

# Hibernate

## 逍遥游记

MASTERING HIBERNATE EASILY

孙卫琴

编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

# 内 容 简 介

Hibernate 是非常流行的对象-关系映射工具。本书由浅入深地介绍运用目前最成熟的 Hibernate 3.3 版本进行 Java 对象持久化的核心技术。本书主要介绍通过 Hibernate API 来访问数据库的方法,还介绍把 Java 对象模型映射到关系数据模型的各种技巧、处理并发问题及实现对话的种种方案。本书将帮助读者编写出具有合理的软件架构,以及好的运行性能和并发性能的实用 Hibernate 应用。书中内容注重理论与实践相结合,列举大量具有典型性和实用价值的 Hibernate 应用实例,并提供详细的开发和部署步骤。

随书配套光盘内容为本书所有范例源程序、本书涉及的软件的最新版本的安装程序。

本书在表述方面,引入了中国传统文化中家喻户晓的《西游记》人物孙悟空,以他学习 Hibernate 为主线,以为花果山实现信息化为案例,带领读者逐步领略 Hibernate 技术的种种神通妙用,大大增加了书的趣味性。只要读者具备了 Java 基础知识,就能轻松阅读本书,快速掌握 Hibernate 技术。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有,侵权必究。

## 图书在版编目(CIP)数据

Hibernate 逍遥游记/孙卫琴编著.-北京:电子工业出版社,2010.7

(Java 开发专家)

ISBN 978-7-121-10967-6

I. ①H… II. ①孙… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2010)第 095549 号

责任编辑:郭 晶 何郑燕

文字编辑:田 蕾

印 刷:北京东光印刷厂

装 订:三河市皇庄路通装订厂

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:787×1092 1/16 印张:21 字数:540.8 千字 彩插:1

印 次:2010 年 7 月第 1 次印刷

印 数:5 000 册 定价:43.80 元(含光盘 1 张)

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn,盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。

“开发专家之 Sun ONE”全新提升为“Java 开发专家”系列  
——源自精品 成就理想

## 出版说明

### ★ 从“开发专家之 Sun ONE”到“Java 开发专家”

“开发专家之 Sun ONE”系列丛书从诞生之日至今，已经九岁了。在这个系列里面，我们一直努力体现着这么一个理念：用一种较为敏锐的视角来跟踪 IT 技术的发展轨迹，并把可能为广大程序员所希望获得的知识，用图书出版的方式奉献给大家。

在这个系列中，我们陆续出版了约 30 种图书，有《Java 与模式》、《JSP 应用开发详解（第二版）》、《精通 EJB（第三版）》、《Tomcat 与 Java Web 应用开发详解》、《精通 Struts：基于 MVC 的 Java Web 设计与开发》、《JBoss 管理与开发核心技术（第三版）》、《精通 Spring》、《精通 Hibernate：Java 对象持久化技术详解》等一大批读者朋友耳熟能详的作品。很多作品都是在国内没有同类图书的情况下出版的。在这几年的出版工作中，我们时刻感受着市场的风险，也时刻收获着无数读者给我们的认可。

在这个系列中，凝聚了大量资深技术专家的心血。有大家都熟知的阎宏、刘晓华、孙卫琴、罗时飞等，还有一些正在不断腾跃的开发高手。这些非常优秀的国内原创作者们一直都在支持着“开发专家之 Sun ONE”系列的出版工作，在这里，我们要向他们说声：谢谢。

桃李不言，下自成蹊。由于这些年“开发专家之 Sun ONE”在“两个效益”中的杰出表现，电子工业出版社授予这个系列“最佳品牌奖”。

时代不断前进，技术不断变革。为了顺应 Java 领域的技术发展态势，为了赋予这个经典的图书系列更强的生命力，我们将“开发专家之 Sun ONE”升级为“Java 开发专家”。我们将继承原有的出版理念，紧密跟踪技术热点和发展趋势，会聚更多优秀作者，全力奉献更经典的作品。

### ★ 规划你的 Java 开发之路

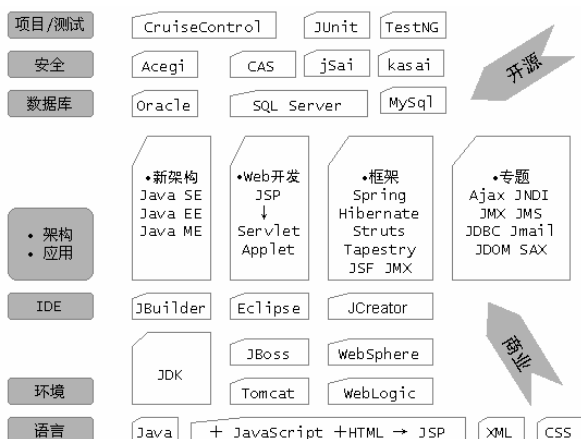
喜马拉雅山脉的最高峰不断地在温室效应中降低，而 Java 世界的颠峰永远都在技术人员的追求中不断升高。每个人都有不同的路，每个人都有不同的行路方式，不过，往往“到了山顶才发现，错误的路和正确的路就差那么几步！”

身处 Java 洪流中的程序员最累（不过大家都说 Java 程序员薪水最高，呵呵），我们简单整理了一下 Java 领域的相关技术、工具、架构，如下图所示。这个框图中的每一个英文单词（或缩写）都可以写成一本书。Java 领域还有一个特点，那就是商业产品和开源产品层出不穷，潮流不断。相比于其他领域，如.NET，Java 开发更是体现了这句谚语：条条大路通罗马。

罗马只有一个，大路却有多条。看上去，似乎到罗马很容易，反正路多嘛。不过，路多却容易迷失方向。当你在 Java 领域中摸爬滚打几年后，发现自己在无数条道路上走了很久，却不知道罗马何日才能到达，甚至连罗马的方向都不知道，这时你肯定会很失落。

很遗憾，在这个简短的出版说明文章里面，我们无法告诉你每一条连贯的、不费周折的通往罗马的道路该如何走。或许，通过“Java 开发专家”系列中的某本书，你可以找到属于你的正确道路。在一般情况下，我们不会就某一项很窄的话题来单独写一本书，我们还是希望通过我们的一些专业和智慧，尽力把一些相关技术整合起来，用较为简明的方式表达出来，最后由你来选择。

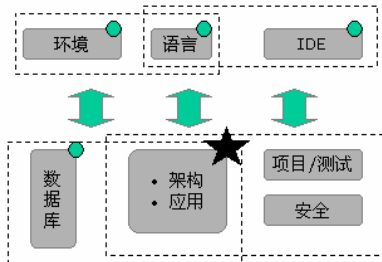
这里有句话与大家共勉：少走弯路，就是捷径！



### ★ “Java 开发专家”的奉献

犹如在上面那个框图中展现的那样，我们希望在各个层面、各个方向上都能给读者奉献出优秀的图书作品，全面体现技术与应用的结合。从宏观上看，我们会从语言、IDE、环境、数据库、架构与应用、安全、项目与测试等方面进行选择，选出一些读者迫切需要的技术来先行规划。

“Java 开发专家”虽然新禧初绽，但因其源自盛放的“开发专家之 Sun ONE”系列而根基稳健，两个系列会有一段很长的并行时间，我们会用一种优化的方式来保证读者的顺利选择。无论哪一个系列，必定都有大家喜欢的图书。



在技术上，有着持久化的方法，在学习上，也需要有持久化的精神。

从“开发专家之 Sun ONE”到“Java 开发专家”，希望可以带给你持久化的动力。

### 联系方式

咨询电话：(010) 68134545 88254160

电子邮件：support@fecit.com.cn

服务网址：<http://www.fecit.com.cn> <http://www.fecit.net>

通用网址：计算机图书、飞思、飞思教育、飞思科技、FECIT

在 Java 领域，访问关系数据库的最原始、最直接的方法是借助 JDBC API。这种方式的优点是运行效率高，缺点是在 Java 程序代码中嵌入大量 SQL 语句，使得项目难以维护。在开发具有分层结构的企业级 Java 应用时，如图 P-1 所示，可以通过 JDBC API 来开发单独的持久化层，把数据库访问操作封装起来，提供简洁的 API，供业务逻辑层统一调用。但是，如果关系数据模型非常复杂，那么直接通过 JDBC API 来实现持久化层需要有专业的知识。对于企业应用的开发人员，花费大量时间从头开发自己的持久化层不是很可行。

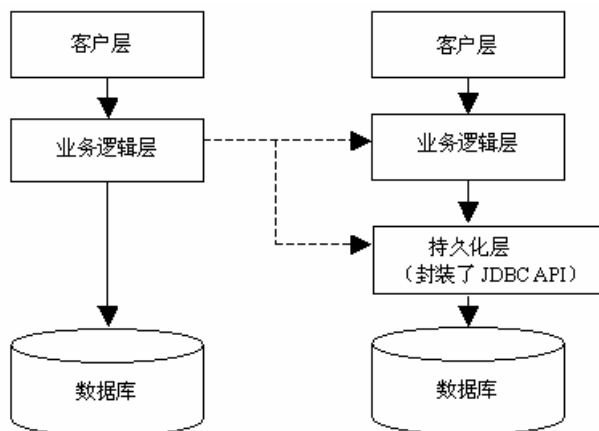


图 P-1 从业务逻辑层分离出单独的持久化层

幸运的是，目前在持久化层已经有好多种现成的持久化软件可供选用，有些是商业性的，如 TopLink；有些是非商业性的，如 JDO 和 Hibernate。Hibernate 是一个基于 Java 的开放源代码的持久化软件，它对 JDBC 做了轻量级封装，不仅提供 ORM（Object Relation Mapping，对象-关系映射）服务，还提供数据查询和数据缓存功能，Java 开发人员可以方便地通过 Hibernate API 来操纵数据库。

现在，越来越多的 Java 开发人员通过 Hibernate 来访问关系数据库，以节省和访问数据库有关的 30% 的 JDBC 编程工作量。

本书结合大量典型的实例，由浅入深地介绍运用目前最成熟的 Hibernate 3.3 版本来访问关系数据库的技术。

本书在表述方面，引入了中国传统文化中家喻户晓的《西游记》人物孙悟空，以他学习 Hibernate 为主线，以为花果山实现信息化为案例，带领读者逐步领略 Hibernate 技术的种种神通妙用，大大增加了书的趣味性。



# 本书的组织结构和主要内容

本书按照由浅入深、前后照应的顺序来安排内容，主要包含以下内容。

## 1. Hibernate入门（第 1 和第 2 章）

第 1 和第 2 章为入门篇。第 1 章概要介绍访问关系数据库的各种途径，通过比较，帮助读者理解通过 Hibernate 来访问数据库的优势。

第 2 章以一个简单的 Hibernate 应用实例——monkeys 应用为例，引导读者把握设计、开发和部署 Hibernate 应用的整体流程，理解 Hibernate 在分层的软件结构中所处的位置。

## 2. 对象-关系映射技术（第 3、4、5、8、9、10、11、12 和第 13 章）

本书重点介绍的内容之一就是如何运用 Hibernate 工具，把对象模型映射到关系数据模型，相关章节包括：

第 3 章：介绍对象-关系映射的基础知识。

第 4 章：介绍对象标识符的映射方法。

第 5 章：介绍一对多关联关系的映射方法。

第 8 章：介绍组成关系的映射方法。

第 9 章：介绍 Java 类型、SQL 类型和 Hibernate 映射类型之间的对应关系。

第 10 章：介绍继承关系的映射方法。

第 11 章：介绍 Java 集合类的用法，这一章主要是为第 12 章做铺垫的。

第 12 章：介绍 Java 集合的映射方法。

第 13 章：介绍一对一和多对多关联关系的映射方法。

## 3. 通过Hibernate API操纵数据库（第 6 和第 16 章）

第 6 章介绍运用 Hibernate API 来保存、更新、删除、加载或查询 Java 对象的方法，并介绍 Java 对象在持久化层的 4 种状态：临时状态、持久化状态、游离状态和删除状态。深入理解 Java 对象的 4 种状态及状态转化机制，是编写健壮的 Hibernate 应用程序的必要条件。

第 16 章介绍 Session 的生命周期的管理方式，以及会话的实现方式。这一章的内容将帮助读者简化 Hibernate 应用的程序代码，并且为应用设计合理的软件架构。

## 4. Hibernate的检索策略和检索方式（第 7 章）

第 7 章介绍 Hibernate 的各种检索策略，对每一种检索策略，都介绍它的适用

场合。合理运用 **Hibernate** 的检索策略及检索技巧，是提高 **Hibernate** 应用性能的重要手段。第 7 章还介绍 **HQL** 查询语句的语法，以及 **QBC API** 的基本使用方法。

## 5. 数据库事务与并发（第 14 和第 15 章）

第 14 章先介绍数据库事务的概念，接着介绍运用 **Hibernate API** 来声明事务边界的方法。

第 15 章介绍在并发环境中出现的各种并发问题，然后介绍采用悲观锁或乐观锁来避免并发问题的方法。

## 本书的范例程序

为了使读者不但能掌握用 **Hibernate** 来持久化 **Java** 对象的理论，并且能迅速获得开发 **Hibernate** 应用的实际经验，完全掌握并会灵活运用 **Hibernate** 技术，本书几乎为每一章都提供了完整的 **Hibernate** 应用范例，在本书配套光盘中包含了所有范例源文件。

为了方便初学者顺利地运行本书的范例，光盘上提供的所有范例程序都是可运行的。读者只要把它们复制到本地机器上，就能够运行，不需要再做额外的配置。此外，在每个范例的根目录下还提供了 **ANT** 工具的工程文件 **build.xml**，它用于编译和运行范例程序。

## 这本书是否适合您

把 **Java** 对象持久化到关系数据库，几乎是所有企业 **Java** 应用必不可少的重要环节，因此本书适用于所有从事开发 **Java** 应用的读者。**Hibernate** 是 **Java** 应用和关系数据库之间的桥梁，阅读本书，要求读者具备 **Java** 语言和关系数据库的基础知识。

实践是掌握 **Hibernate** 的好方法。为了让读者彻底掌握并学会灵活运用 **Hibernate**，本书为每一章都提供了典型的范例，在本书配套光盘上提供了完整的源代码，以及软件安装程序。建议读者在学习 **Hibernate** 技术的过程中，善于将理论与实践相结合，达到事半功倍的效果。

本书与作者的另一本书《精通 **Hibernate**：Java 对象持久化技术详解》一书相比，前者的特色在于化繁为简，以通俗浅显的语言介绍 **Hibernate** 的主要技术及对象-关系映射的核心思想。后者则更为详细全面地阐述 **Hibernate** 的各种技术，并且介绍运用 **Hibernate** 来开发项目的高级技巧和性能优化的细节。前者提纲挈领，后

者包罗万象，两者各有千秋，相得益彰。

## 光盘使用说明

本书配套光盘包含以下目录。

### 1. software目录

在该目录下包含了本书内容涉及的所有软件的最新版本的安装程序，包括：

- (1) Hibernate 核心软件包 (Hibernate 3.3)。
- (2) Hibernate 工具软件包 (HibernateTools 3.2)。
- (3) MySQL 服务器的安装软件 (MySQL 5)。
- (4) MySQL 的 JDBC 驱动程序 (Mysql-Connector-Java-3.1.7)
- (5) ANT 的安装软件 (Ant 1.7)。

### 2. sourcecode目录

在该目录下提供了本书所有的源程序。

本书在编写过程中得到了飞思数字创意出版中心、JavaThinker.org 网站的支持与帮助，在此表示衷心的感谢！参与编写的人员有孙卫琴、李洪成、曹文伟、李红军、孙定金、曹汉玉、张玲、吴厚鹏、刘琨、赵鹏、赵明、张黎平、刘巧云、李平安、王秀龄等十五人。尽管我们尽了最大努力，但本书难免会有不妥之处，欢迎各界专家和读者朋友批评指正。以下网址是作者为本书提供的技术支持网址，读者可通过它下载与本书相关的资源（如源代码、软件安装程序和讲义等），还可以与其他读者交流学习心得，以及对本书提出宝贵意见：

[http://www.javathinker.org/hibernate\\_taste.jsp](http://www.javathinker.org/hibernate_taste.jsp)





第 1 章	访问关系数据库的途径 .....	1
1.1	创建关系数据库表 .....	1
1.2	访问关系数据库的途径 .....	2
1.2.1	通过数据库的自带客户程序与数据库服务器交互 .....	2
1.2.2	通过 Java 程序与数据库服务器交互 .....	3
1.3	Java 程序通过 JDBC API 访问数据库 .....	6
1.4	Java 程序通过 Hibernate API 访问数据库 .....	9
1.5	Java 对象的持久化概念 .....	12
1.6	小结 .....	13
第 2 章	第一个 Hibernate 应用 .....	15
2.1	创建 Hibernate 的配置文件 .....	15
2.2	创建持久化类 .....	16
2.3	创建数据库 Schema .....	18
2.4	创建对象-关系映射文件 .....	18
2.5	通过 Hibernate API 操纵数据库 .....	21
2.5.1	Hibernate 的初始化 .....	24
2.5.2	访问 Hibernate 的 Session 接口 .....	25
2.6	运行 monkeys 应用 .....	28
2.6.1	创建运行本书范例的系统环境 .....	28
2.6.2	创建 monkeys 应用的目录结构 .....	32
2.6.3	运行 monkeys 应用 .....	33
2.6.4	给 monkeys 应用加入用户界面 .....	36
2.7	小结 .....	36
第 3 章	对象-关系映射基础 .....	39
3.1	持久化类的属性及访问方法 .....	39
3.1.1	基本类型属性和包装类型属性 .....	40
3.1.2	Hibernate 访问持久化类属性的策略 .....	42
3.1.3	在持久化类的访问方法中加入程序逻辑 .....	42
3.1.4	设置派生属性 .....	44
3.1.5	控制 insert 和 update 语句 .....	45
3.2	处理 SQL 引用标识符 .....	46
3.3	使用 XML 格式的配置文件 .....	47
3.4	运行本章的范例程序 .....	48
3.5	小结 .....	54

# Contents

第 4 章	映射对象标识符 .....	57
4.1	关系数据库按主键区分不同的记录 .....	58
4.1.1	把主键定义为自动增长标识符类型 .....	59
4.1.2	从序列 (Sequence) 中获取自动增长的标识符 .....	59
4.2	Java 语言按内存地址区分不同的对象 .....	60
4.3	Hibernate 用对象标识符 (OID) 来区分对象 .....	61
4.4	Hibernate 的内置标识符生成器的用法 .....	64
4.4.1	increment 标识符生成器 .....	68
4.4.2	identity 标识符生成器 .....	70
4.4.3	sequence 标识符生成器 .....	71
4.4.4	hilo 标识符生成器 .....	73
4.4.5	native 标识符生成器 .....	74
4.5	映射自然主键 .....	76
4.6	小结 .....	77
第 5 章	映射一对多关联关系 .....	79
5.1	建立多对一的单向关联关系 .....	81
5.1.1	关于 TransientObjectException 异常 .....	86
5.1.2	级联保存和更新 .....	87
5.2	映射一对多双向关联关系 .....	88
5.2.1	<set>元素的 inverse 属性 .....	93
5.2.2	级联删除 .....	96
5.2.3	父子关系 .....	97
5.3	小结 .....	98
第 6 章	通过 Hibernate 操纵对象 .....	99
6.1	理解 Session 的缓存 .....	99
6.1.1	Session 的缓存的作用 .....	101
6.1.2	脏检查及清理缓存的机制 .....	103
6.2	Java 对象在 Hibernate 持久化层的状态 .....	105
6.2.1	临时对象的特征 .....	107
6.2.2	持久化对象的特征 .....	107
6.2.3	被删除对象的特征 .....	108
6.2.4	游离对象的特征 .....	109
6.3	Session 接口的详细用法 .....	109
6.3.1	Session 的 save() 方法 .....	110
6.3.2	Session 的 load() 和 get() 方法 .....	111
6.3.3	Session 的 update() 方法 .....	112
6.3.4	Session 的 saveOrUpdate() 方法 .....	115
6.3.5	Session 的 merge() 方法 .....	116

6.3.6	Session 的 delete()方法 .....	117
6.4	级联操纵对象图 .....	118
6.5	批量处理数据 .....	120
6.5.1	通过 Session 来进行批量操作 .....	121
6.5.2	通过 StatelessSession 来进行批量操作 .....	123
6.5.3	通过 HQL 来进行批量操作 .....	124
6.6	Hibernate 的二级缓存结构 .....	125
6.7	小结 .....	126
第 7 章	Hibernate 的检索策略和检索方式 .....	129
7.1	Hibernate 的检索策略 .....	131
7.1.1	类级别的检索策略 .....	133
7.1.2	一对多和多对多关联的检索策略 .....	136
7.1.3	多对一和一对一关联的检索策略 .....	139
7.1.4	在应用程序中显式指定迫切左外连接检索策略 .....	143
7.1.5	比较 3 种检索策略 .....	143
7.2	检索方式 .....	145
7.2.1	HQL 检索方式 .....	145
7.2.2	QBC 检索方式 .....	146
7.2.3	SQL 检索方式 .....	147
7.3	小结 .....	148
第 8 章	映射组成关系 .....	149
8.1	建立精粒度对象模型 .....	150
8.2	建立粗粒度关系数据模型 .....	151
8.3	映射组成关系 .....	152
8.3.1	区分值 (Value) 类型和实体 (Entity) 类型 .....	155
8.3.2	在应用程序中访问具有组成关系的持久化类 .....	156
8.4	映射复合组成关系 .....	160
8.5	小结 .....	161
第 9 章	Hibernate 的映射类型 .....	163
9.1	Hibernate 的内置映射类型 .....	163
9.1.1	Java 基本类型的 Hibernate 映射类型 .....	163
9.1.2	Java 时间和日期类型的 Hibernate 映射类型 .....	164
9.1.3	Java 大对象类型的 Hibernate 映射类型 .....	164
9.1.4	JDK 自带的个别 Java 类的 Hibernate 映射类型 .....	165
9.1.5	使用 Hibernate 内置映射类型 .....	165
9.2	客户化映射类型 .....	166
9.3	用客户化映射类型取代 Hibernate 组件 .....	171
9.4	运行范例程序 .....	175

## Contents

9.5	小结 .....	181
第 10 章	映射继承关系 .....	183
10.1	继承关系树的每个具体类对应一个表 .....	184
10.1.1	创建映射文件 .....	185
10.1.2	操纵持久化对象 .....	187
10.2	继承关系树的根类对应一个表 .....	190
10.2.1	创建映射文件 .....	190
10.2.2	操纵持久化对象 .....	192
10.3	继承关系树的每个类对应一个表 .....	194
10.3.1	创建映射文件 .....	195
10.3.2	操纵持久化对象 .....	197
10.4	选择继承关系的映射方式 .....	199
10.5	小结 .....	200
第 11 章	Java 集合 .....	203
11.1	Set (集) .....	204
11.1.1	Set 的一般用法 .....	204
11.1.2	HashSet 类 .....	206
11.1.3	TreeSet 类 .....	207
11.2	List (列表) .....	212
11.3	Map (映射) .....	213
11.4	小结 .....	214
第 12 章	映射值类型集合 .....	217
12.1	映射 Set (集) .....	217
12.2	映射 Bag (包) .....	221
12.3	映射 List (列表) .....	224
12.4	映射 Map .....	227
12.5	对集合排序 .....	230
12.5.1	在数据库中对集合排序 .....	231
12.5.2	在内存中对集合排序 .....	232
12.6	小结 .....	236
第 13 章	映射实体关联关系 .....	239
13.1	映射一对一关联 .....	239
13.1.1	按照外键映射 .....	240
13.1.2	按照主键映射 .....	245
13.2	映射单向多对多关联 .....	248
13.3	映射双向多对多关联关系 .....	251
13.3.1	关联两端使用 <set> 元素 .....	252
13.3.2	使用组件类集合 .....	253

13.3.3	把多对多关联分解为两个一对多关联 .....	258
13.4	小结 .....	260
第 14 章	声明数据库事务 .....	261
14.1	数据库事务的概念 .....	261
14.2	声明事务边界的方式 .....	263
14.3	在 mysql.exe 程序中声明事务 .....	265
14.4	Java 应用通过 JDBC API 声明事务 .....	267
14.5	Java 应用通过 Hibernate API 声明事务 .....	269
14.5.1	处理异常 .....	270
14.5.2	Session 与事务的关系 .....	271
14.5.3	设定事务超时 .....	274
14.6	小结 .....	274
第 15 章	处理并发问题 .....	275
15.1	多个事务并发运行时的并发问题 .....	275
15.1.1	第一类丢失更新 .....	276
15.1.2	脏读 .....	277
15.1.3	虚读 .....	277
15.1.4	不可重复读 .....	278
15.1.5	第二类丢失更新 .....	279
15.2	数据库系统的锁的基本原理 .....	279
15.3	数据库的事务隔离级别 .....	280
15.3.1	在 mysql.exe 程序中设置隔离级别 .....	282
15.3.2	在应用程序中设置隔离级别 .....	282
15.4	在应用程序中采用悲观锁 .....	283
15.5	利用 Hibernate 的版本控制来实现乐观锁 .....	289
15.5.1	使用<version>元素 .....	289
15.5.2	使用<timestamp>元素 .....	295
15.5.3	对游离对象进行版本检查 .....	296
15.6	实现乐观锁的其他方法 .....	297
15.7	小结 .....	298
第 16 章	管理 Session 和实现对话 .....	301
16.1	Hibernate 管理 Session 对象的方式 .....	302
16.2	Session 对象的生命周期与本地线程绑定 .....	304
16.3	实现对话 .....	307
16.3.1	使用游离对象 .....	309
16.3.2	使用手工清理缓存模式下的 Session .....	312
16.4	Hibernate 委托程序来管理 Session .....	314
16.5	小结 .....	317

# 第 1 章 访问关系数据库的途径

且说孙悟空帮助唐僧到西天取到真经后，就在天上逍遥自在地当起了斗战胜佛。不知过了多少年头，有一天，他到人间游玩，发现大部分人无论上班或生活，都要摆弄一个名叫“电脑”的玩艺。悟空聪明好学，很快就掌握了玩电脑的本领。不仅如此，他还把兴趣转向软件开发领域，如今，他已熟悉面向对象的 Java 编程语言，了解关系数据库的基础知识，还熟悉用于直接和关系数据库交流的 SQL 语言。

悟空掌握了这些软件开发的基础知识后，首先想到了为他的老家花果山干点实事。他要把花果山的猴子们的信息全部存入数据库，让花果山也与时俱进，进入信息时代。

在本章，悟空尝试了各种访问关系数据库的途径。经过权衡利弊后，悟空最终决定在 Java 程序中借助 Hibernate 来完成操纵数据库的重任。

## 1.1 创建关系数据库表

悟空在他所熟悉的 MySQL 关系数据库中创建了一张名为“MONKEYS”的表，表的结构如图 1-1 所示。

MONKEYS 表用来存放猴子的基本信息，比如名字（NAME 字段）、年龄（AGE 字段）和性别（GENDER 字段）。

MONKEYS 表
ID <<PK>>
NAME
AGE
GENDER

图 1-1 MONKEYS 表的结构

MONKEYS 表中的 ID 字段为主键（Primary Key，简称 PK）。数据库表通过主键来保证每条记录的唯一性，每条记录的 ID 值都是唯一的。表的主键最好不具有任何业务含义，即不代表特定业务领域的某种信息。任何有业务含义的字段都有可能随着业务需求的变化而被改变。关系数据库学的最重要的理论之一就是：不要给主键赋予任何业务含义，这样可以提高数据库系统的可维护性。

假如主键具有了业务含义，会出现什么情况呢？以 MONKEYS 表为例，假定把 NAME 字段作为主键，这是一个具有业务含义的主键。假定一开始用户的业务需求为：NAME 字段为 6 位字符串，过了一年后，用户改变了业务需求，规定 NAME 字段为 8 位字符串。当业务需求改变后，就必须修改 MONKEYS 表中所有记录的 NAME 主键的值，此外，对于那些参照 MONKEYS 表，并且把 NAME 字段作为外键的所有其他表，也需要修改表中所有记录的 NAME 外键的值。由此可见，主键

如果具有业务含义，那么即使业务含义发生很小的变化，也可能会给数据库系统造成极大的维护上的开销。

为了使表的主键不具有任何业务含义，一种比较常用的解决方法是使用代理主键，例如，为表定义一个不具有任何业务含义的 ID 字段（也可以叫其他的名字），专门作为表的主键。

## 1.2 访问关系数据库的途径

悟空把 MONKEYS 表创建好以后，就急于把花果山的一个聪明伶俐的小猴的信息存入表中。该小猴的信息如下：

名字：智多星  
年龄：1 岁  
性别：公 (Male)

如图 1-2 所示，悟空看着运行中的 MySQL 数据库服务器，幽默地用猴语对服务器大喊：“嗨，哥们，帮我把这小猴崽的信息记下来。”可惜服务器无动于衷，它哪里懂得猴语啊。悟空宽宏大量地对服务器笑笑，经过西天取经的磨练，悟空已经改了动不动就亮出金箍棒唬人的毛病。他明白，要与对方交流，就首先要熟悉对方的语言。假如双方不得不各自使用不同的语言，就得找个合适的翻译员。

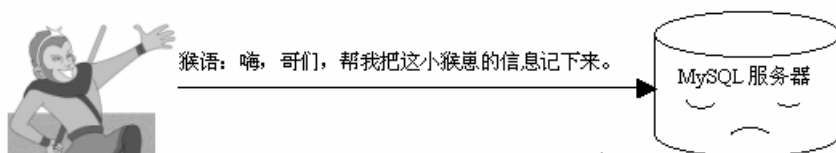


图 1-2 悟空试图用猴语和数据库交流

数据库服务器只懂得 SQL 语言，因此悟空必须想办法把自己的请求用 SQL 语言表达出来，然后通过特定的客户程序把 SQL 语句传给服务器，服务器才会执行这些 SQL 语句。

### 1.2.1 通过数据库的自带客户程序与数据库服务器交互

悟空打开了 MySQL 服务器的自带客户程序(对应 MySQL 安装目录的 bin 子目录下的 mysql.exe 程序)，然后在电脑键盘上霹雳啪啦一阵敲打，他向 MySQL 自带客户程序输入了一些 SQL 语句，参见图 1-3。



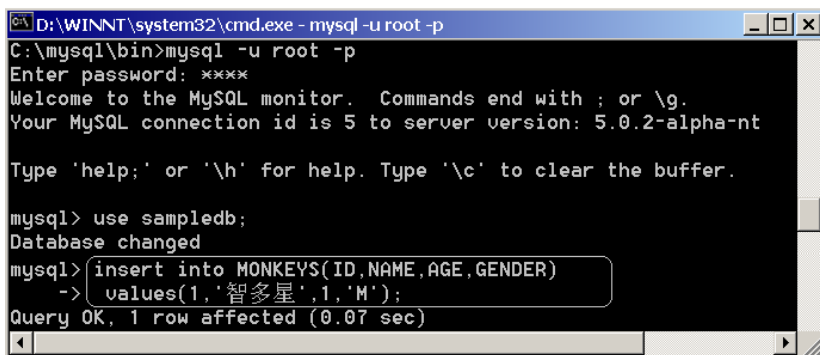


图 1-3 MySQL 自带客户程序 (mysql.exe) 的命令行界面

在图 1-3 中，悟空输入的 SQL 语句为：

```
insert into MONKEYS(ID,NAME,AGE,GENDER)
values(1,'智多星',1,'M');
```

MySQL 自带客户程序接收到悟空输入的 SQL 语句后，就会把它如实转交给 MySQL 服务器，这正是 MySQL 服务器所能理解的语言，这一回，服务器乖乖地听从悟空的吩咐，把小猴崽的信息保存到 MONKEYS 表中，参见图 1-4。



图 1-4 用户通过 MySQL 自带客户程序与 MySQL 服务器通信

### 1.2.2 通过Java程序与数据库服务器交互

尽管直接通过 MySQL 的自带客户程序来访问 MySQL 数据库服务器也是可行的，但是实际操作起来却很麻烦。悟空打算让刚参加了电脑扫盲班的小不点来管理猴子信息。小不点不太懂 SQL 语言，要让他直接操纵 MySQL 自带客户程序，有很大风险，万一他不小心输错了 SQL 语句，可能会误删除或误修改数据库中的重要数据。所以必须给小不点定制专门的客户程序，最好有更加直观的界面，让小不点容易操作，而且不觉得枯燥。

悟空刚刚学会了 Java 编程语言，何不用 Java 程序来编写一个访问数据库的客户程序呢，参见图 1-5。



图 1-5 用户通过 Java 程序与 MySQL 服务器通信

悟空先编写了一个代表猴子信息的 `Monkey` 类，参见例程 1-1。

#### 例程 1-1 `Monkey.java`

```

package mypack;

public class Monkey{
    private Long id;
    private String name;
    private int age;
    private char gender;

    public Monkey(){}

    public Long getId(){
        return id;
    }

    private void setId(Long id){
        this.id = id;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name=name;
    }

    public int getAge(){
        return age;
    }

    public void setAge(int age){
        this.age =age ;
    }

    public char getGender(){
        return gender;
    }

    public void setGender(char gender){

```

```

        this.gender =gender ;
    }
}

```

以上 Monkey 类包含一些属性，以及与之对应的 getXXX()和 setXXX()方法。按照这种风格编写出来的 Java 类也叫做 JavaBean。

悟空接着利用 Java Swing 组件，编写了一个用于保存猴子信息的界面程序 MonkeyGui 类。运行 MonkeyGui 类，就会出现一个很直观而且容易操作的界面，参见图 1-6。



图 1-6 MonkeyGui 类的界面

尽管界面程序很快编出来了，悟空还是在最关键的一步上卡住了。当小不点按下界面上的【保存】按钮时，到底如何把小不点输入的猴子信息保存到数据库中呢？悟空暂且虚晃一枪，定义了一个 BusinessService 类，拟定由 BusinessService 类的 saveMonkey()方法来完成这个重任。MonkeyGui 界面上的【保存】按钮的 ActionListener 监听器监听到小不点按下按钮的事件后，就会在它的 actionPerformed()方法中调用 BusinessService 类的 saveMonkey()方法：

```

public void actionPerformed(ActionEvent event){
    try{
        Monkey monkey=new Monkey();
        //根据用户界面上输入的数据来设置 Monkey 对象的信息
        monkey.setName(nameTf.getText().trim());
        monkey.setAge(Integer.parseInt(ageTf.getText().trim()));
        monkey.setGender(genderTf.getText().trim().charAt(0));

        //保存 Monkey 对象
        businessService.saveMonkey(monkey);
        logLb.setText("猴子信息已经保存成功。");
    }catch(Exception e){
        logLb.setText("猴子信息保存失败。");
        e.printStackTrace();
    }
}

```

至于到底如何实现 BusinessService 类的 saveMonkey()方法，悟空暂且还没谱，还得先翻阅一些 Java 资料，再现学现用呢。在下面的 1.3 节，将介绍悟空学以致用，解决图 1-5 中 Java 程序与关系数据库通信的问题。

## 1.3 Java程序通过JDBC API访问数据库

Java 程序使用的是 Java 语言，而关系数据库只懂 SQL 语言，要让使用不同语言的双方成功交流，就得请个翻译员了。这个翻译员就是 JDBC（Java Database Connectivity，Java 数据库连接器）驱动程序。如图 1-7 所示，JDBC 驱动程序是连接 Java 程序和关系数据库的桥梁，只有通过它，Java 程序才能与关系数据库通信。许多数据库服务器厂商为了让 Java 程序能够访问它们的数据库，都提供了相应的 JDBC 驱动程序。例如，从 MySQL 的官方网站（[www.mysql.com](http://www.mysql.com)）上就可以下载到 MySQL 的 JDBC 驱动程序类库。

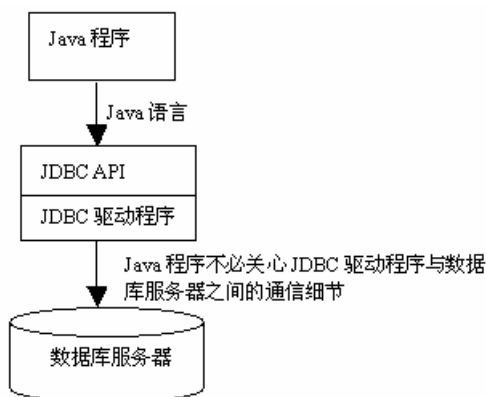


图 1-7 Java 程序通过 JDBC 驱动程序与数据库服务器通信

从图 1-7 可以看出，有了 JDBC 驱动程序，Java 程序就只需和 JDBC 驱动程序的 API 交互了。至于 JDBC 驱动程序如何与数据库服务器通信，取决于 JDBC 驱动程序的具体实现，Java 程序无须了解它们的通信细节。JDBC API 使用的是面向对象的 Java 语言，它为 Java 程序提供了用于访问数据库的接口和类，主要包括：

- **DriverManager**：代表驱动程序管理器，负责创建数据库连接。
- **Connection**：代表数据库连接。
- **Statement**：负责执行 SQL 语句。
- **PreparedStatement**：负责执行 SQL 语句，具有预定义 SQL 语句的功能。
- **ResultSet**：代表 SQL 查询语句的查询结果集。

图 1-8 显示了这些类的关系。

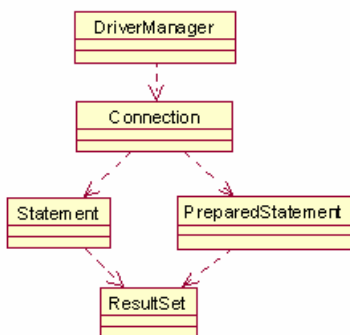


图 1-8 JDBC API 中主要类的类框图

悟空快速掌握了 JDBC API 中的主要接口和类的用法。在 Java 程序中，通过 JDBC API 访问数据库包含以下基本步骤。

(1) 获得要访问的数据库的 JDBC 驱动程序的类库文件，把它放到 classpath 中。

(2) 在程序中加载并注册 JDBC 驱动程序。例如，以下代码用于加载并注册 MySQL 驱动程序：

```
//加载 MySQL Driver 类
Class.forName("com.mysql.jdbc.Driver");
//注册 MySQL Driver
java.sql.DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

(3) 建立与数据库的连接：

```
Connection con = java.sql.DriverManager.getConnection
(dburl,user,password);
```

getConnection()方法中有 3 个参数，dburl 表示连接数据库的 JDBC URL，user 和 password 分别表示连接数据库的用户名和口令。

JDBC URL 的一般形式为：

```
jdbc:drivertype:driversubtype://parameters
```

drivertype 表示驱动程序的类型。driversubtype 是可选的参数，表示驱动程序的子类型。parameters 通常用来设定数据库服务器的 IP 地址、端口号和数据库的名称。对于 MySQL 数据库连接，JDBC URL 采用如下形式：

```
jdbc:mysql://localhost:3306/SAMPLEDB
```

(4) 创建 Statement 对象，准备执行 SQL 语句：

```
Statement stmt = con.createStatement();
```

(5) 执行 SQL 语句：

```
String sql="insert into MONKEYS(ID,NAME,AGE,GENDER) "
        +"values(1,'智多星',1,'M')";
stmt.executeUpdate(sql); //执行 SQL 语句
```

(6) 依次关闭 Statement 和 Connection 对象:

```
stmt.close();
con.close();
```

悟空编写的 BusinessService 类的源代码参见例程 1-2。

### 例程 1-2 通过 JDBC API 来访问数据库的 BusinessService.java

```
package mypack;
import java.io.*;
import java.util.*;
import java.sql.*;

public class BusinessService{
    private String dbUrl="jdbc:mysql://localhost:3306/SAMPLEDB";
    private String dbUser="root";
    private String dbPwd="1234";

    static{
        try{
            Class.forName("com.mysql.jdbc.Driver");
            DriverManager.registerDriver(new com.mysql.jdbc.Driver());
        }catch(Exception e){throw new RuntimeException(e);}
    }

    /** 持久化一个 Monkey 对象 */
    public void saveMonkey(Monkey monkey){
        Connection con=null;
        try {
            //建立数据库连接
            con = java.sql.DriverManager.getConnection(dbUrl,dbUser,dbPwd);
            //创建一个 SQL 声明
            Statement stmt = con.createStatement();
            //向 MONKEYS 表插入记录
            stmt.executeUpdate("insert into MONKEYS(NAME,AGE,GENDER)
                values( "
                +"'" +monkey.getName()+"',"
                +monkey.getAge()+" ,"
                +"'" +monkey.getGender()+"'")");
            stmt.close();
        }catch(Exception e) {
            throw new RuntimeException(e);
        } finally {
            try{
```

```

        if(con!=null)con.close();
    }catch(Exception e){e.printStackTrace();}
    }
}

```

由于 JDBC API 的用法不是本书介绍的重点，因此不对 `BusinessService` 类的代码做详细介绍。

再次运行 `MonkeyGui` 类，`MonkeyGui` 类就可以通过 `BusinessService` 类的 `saveMonkey()` 方法来保存猴子信息了。参见图 1-9。

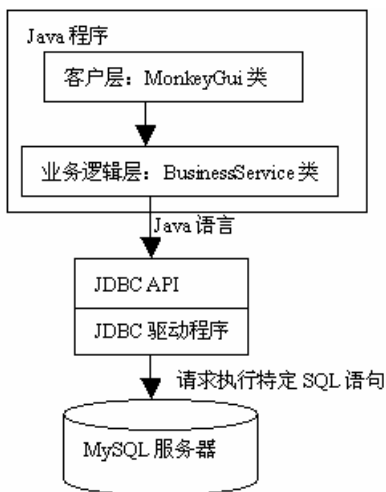


图 1-9 MonkeyGui 类通过 BusinessService 类来保存猴子信息

在图 1-9 中，Java 程序针对数据库服务器而言，它是数据库服务器的客户程序。而 Java 程序本身还可以分为客户层和业务逻辑层。在本例中，`MonkeyGui` 类属于客户层，提供图形用户界面；而 `BusinessService` 类属于业务逻辑层，负责执行业务逻辑及访问数据库。

## 1.4 Java程序通过Hibernate API访问数据库

悟空掌握了 JDBC API 的用法后，洋洋得意了一阵，觉得自己更加神通广大了。但是过了一阵，它发现用 JDBC API 来访问数据库时，以 1.3 节的例程 1-2 的 `BusinessService` 类的 `saveMonkey()` 方法为例，由于在程序代码中嵌入了 SQL 语句，会导致以下弊端：

- 编程人员必须既懂 Java 语言，又懂 SQL 语言，才能编写数据库访问代码。



- 程序代码中嵌入大量字符串形式的 SQL 语句，降低了程序代码的可读性。
- 程序代码与关系数据库结构绑定在一起，削弱了程序代码的独立性和可维护性。例如，当数据库中 MONKEYS 表的结构发生了改变，比如修改了一个字段的名称，那么程序代码中所有涉及 MONKEYS 表的 SQL 语句也要做相应修改。
- 当编程人员试图向数据库存入一个 Monkey 对象时，他需要了解与 Monkey 对象对应的表为 MONKEYS 表，与 Monkey 对象的 name 属性对应的字段为 NAME 字段，依次类推。因此编程人员必须即熟悉对象模型，又熟悉关系数据模型，还要了解两者的对应关系。所以编程人员不能完全按照面向对象的思维来编写程序代码。

为了解决以上弊端，悟空引进了一种目前很流行的工具软件，名为 **Hibernate**。**Hibernate** 本身也完全用 Java 语言编写出来，并且它对 JDBC API 进行了封装，能提供更加面向对象的数据库访问 API，参见图 1-10。

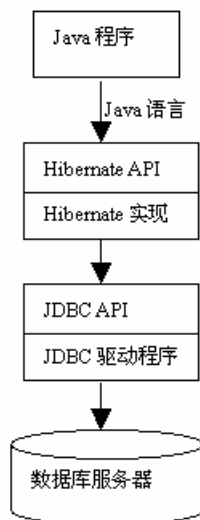


图 1-10 Java 程序通过 Hibernate API 来访问数据库

**Hibernate API** 中最核心的一个接口就是 **Session** 接口，通过 **Session** 接口来向数据库存入一个 **Monkey** 对象非常简单，只要调用它的 `save()` 方法就行了：

```
//嗨，Session 总管，拜托帮我把一个 Monkey 对象保存到数据库中。
session.save(monkey);
```

以上程序代码多么简洁呀，程序代码中不涉及任何 SQL 语句，因此是纯粹的面向对象的程序代码。程序代码只需轻轻松松地向 **Session** 接口发送一条消息：“嗨，Session 总管，拜托帮我把一个 **Monkey** 对象保存到数据库中”，至于到底如何把一个 **Monkey** 对象保存到数据库中，就完全由 **Hibernate** 来代劳了。由此可见，**Hibernate API**

与 JDBC API 相比,前者是更加面向对象的数据库访问 API。本书第 2 章会改写本章 1.3 节的例程 1-2 的 `BusinessService` 类,让它通过 `Hibernate` API 来访问数据库。

`Hibernate` 的实现细节不是本书介绍的重点,本书主要介绍如何使用 `Hibernate`。不过,如果想娴熟地使用 `Hibernate`,最好对它的实现细节稍微有所了解。大致说来,`Session` 接口的实现类的 `save()` 方法将执行以下步骤:

(1) 运用 Java 反射机制,了解到 `Monkey` 对象的类型为 `Monkey.class`。

(2) 参考对象-关系映射元数据 (`Object-Relation Mapping Meta Data`),了解到与 `Monkey` 类对应的表为 `MONKEYS` 表,与 `Monkey` 类的 `name` 属性对应的字段为 `NAME` 字段,依次类推。

(3) 根据以上映射信息,生成 SQL 语句:

```
insert into MONKEYS(ID,NAME,AGE,GENDER)
values(1,'智多星',1,'M');
```

(4) 通过 JDBC API 来执行以上 SQL 语句。

由此可见,`Hibernate` 能够把一个 `Monkey` 对象保存到关系数据库中,把 `Monkey` 对象映射成 `MONKEYS` 表中的一条记录。

在以上步骤 2 中提到了对象-关系映射元数据,它是 `Hibernate` 为自己开小灶,要求 Java 程序必须为自己提供的额外信息,采用 XML 格式。对象-关系映射元数据存放在对象-关系映射文件中,以 `hbm.xml` 作为文件扩展名。如以下例程 1-3 的 `Monkey.hbm.xml` 文件指定了 `Monkey` 类与 `MONKEYS` 表的映射关系。

例程 1-3 `Monkey.hbm.xml`

```
<hibernate-mapping>
  <class name="mypack.Monkey" table="MONKEYS">

    <id name="id" column="ID" type="long">
      <generator class="increment"/>
    </id>

    <property name="name" column="NAME" type="string" not-null=
      "true" />
    <property name="age" column="AGE" type="int" />
    <property name="gender" column="GENDER" type="character"/>

  </class>
</hibernate-mapping>
```

只要提供了用于映射 `Monkey` 类与 `MONKEYS` 表的映射文件，Hibernate 在运行时就能参照映射文件的信息，把 `Monkey` 对象保存到数据库中的 `MONKEYS` 表中。

由此可见，如果直接通过 JDBC API 访问数据库，那么程序中的代码及 SQL 语句就包含了对象模型与关系数据模型之间的映射关系。如果通过 Hibernate API 访问数据库，SQL 语句就从程序代码中分离出去了，不过，应用程序还需为 Hibernate 额外提供 XML 格式的对象-关系映射文件。

要驾驭 Hibernate 工具，不仅要熟悉它的 API 中的接口和类的用法，还要掌握如何进行对象-关系映射，从而为 Hibernate 提供正确的对象-关系映射文件。

## 1.5 Java对象的持久化概念

花果山猴子们的信息作为特定应用领域里的业务数据，有两种表现形式：

- 在内存中表现为 `Monkey` 对象。
- 在关系数据库中表现为 `MONKEYS` 表中的记录。

当 Java 程序在内存中创建了一个 `Monkey` 对象后，它不可能永远存在。最后，它要么从内存中清除，要么被持久化到数据库中。内存无法永久地保存数据，因此必须对 `Monkey` 对象进行持久化。否则，如果 `Monkey` 对象没有被持久化，那么用户在应用程序运行时创建的猴子信息将在应用程序结束运行后随之消失。而一旦 `Monkey` 对象被持久化，它就可以在应用程序再次运行时被重新加载到内存，并重新构造出 `Monkey` 对象。图 1-11 显示了 `Monkey` 对象的持久化过程。

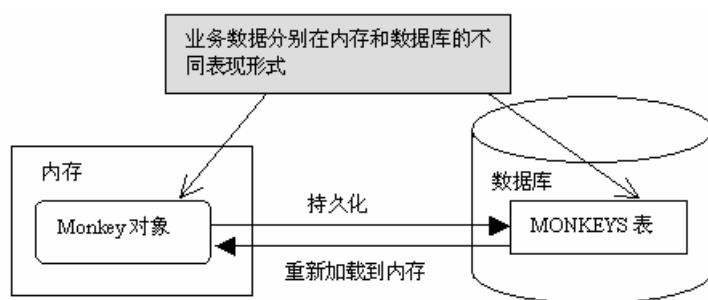


图 1-11 对象的持久化

### Tips

Hibernate 的英文原意是冬眠，冬眠与持久化之间有什么关系呢？Java 对象存在于内存中，Hibernate 能够把 Java 对象永久保存到关系数据库中。形象地理解，可以说 Hibernate 能够让内存中的 Java 对象在关系数据库中“冬眠”。

狭义的理解，“持久化”仅仅指把对象永久保存到数据库中；广义的理解，“持久化”包括和数据库相关的各种操作：

- 保存：把对象永久保存到数据库中。
- 更新：更新数据库中对象的状态。
- 删除：从数据库中删除一个对象。
- 加载：根据特定的 OID（Object Identifier，对象标识符），把一个对象从数据库加载到内存中。
- 查询：根据特定的查询条件，把符合查询条件的一个或多个对象从数据库加载到内存中。

确切地说，数据库中存放的是关系数据，而不是对象。但本书常常出现“从数据库中加载对象”、“删除数据库中的对象”，以及“更新数据库中的对象”等说法。这主要是站在 **Hibernate** 的客户程序端的角度来看待数据库访问操作的。**Hibernate** 封装了数据库访问细节，为客户程序提供了面向对象的持久化语义。客户程序可以假想数据库中存放的就是对象，只需委托 **Hibernate** 从数据库中加载对象、删除对象，以及更新对象就行了，至于 **Hibernate** 如何把这些对象映射为数据库中的相应关系数据，这就属于 **Hibernate** 的分内之事了。

## 1.6 小结

关系数据库能够直接理解的是 **SQL** 语言。访问关系数据库，实际上就是请求数据库执行特定的 **SQL** 语句。本章介绍了访问关系数据库的各种途径，参见图 1-12。

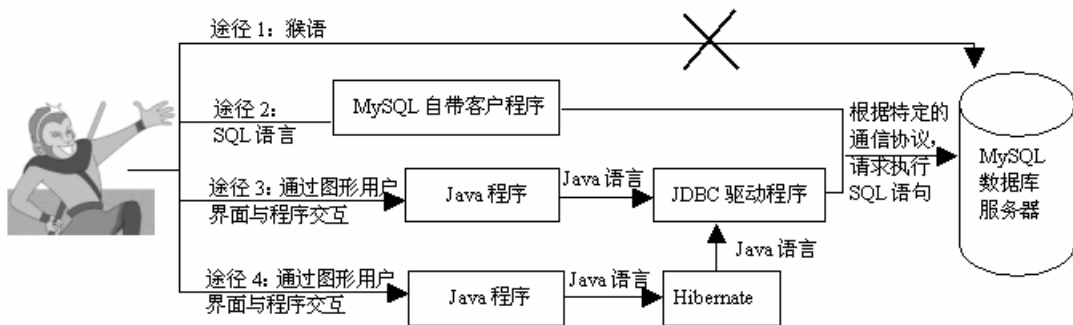


图 1-12 访问关系数据库的各种途径

Java 程序需要直接或间接通过 **JDBC** 驱动程序来和关系数据库通信。**JDBC** 驱动程序是连接 Java 程序和关系数据库的桥梁。当 Java 程序直接通过 **JDBC** 驱动程

序访问数据库时，Java 程序只需和 JDBC API 交互，例如，以下 Java 程序代码通过 JDBC API 来向数据库保存猴子信息：

```
Statement stmt=connection.createStatement();
String sql="insert into MONKEYS(ID,NAME,AGE,GENDER) "
        +"values(1,'智多星',1,'M')";
stmt.executeUpdate(sql); //执行 SQL 语句
```

以上程序代码中必须嵌入 SQL 语句，这使得 Java 程序与关系数据库绑定在一起，削弱了 Java 程序的独立性。而且编程人员不能完全按照面向对象的思维来编写访问数据库的代码，他必须了解数据库结构，熟悉 SQL 语言，才能编写出上述程序代码。

为了解除 Java 程序与关系数据库绑定关系，并且能使得编程人员完全按照面向对象的思维来编写访问数据库的代码，就必须想办法把 SQL 语句从 Java 程序中分离出去。Java 程序借助 Hibernate，就能解决这个问题。Hibernate 通过专门的 XML 格式的对象-关系映射文件来描述对象模型和关系数据库模型之间的对应关系，而 Java 程序代码本身无须提供任何关于关系数据库的结构信息。而且 Hibernate 封装了 JDBC API，为 Java 程序提供了更加面向对象的数据库访问 API。例如以下 Java 程序代码通过 Hibernate API 中的 Session 接口来向数据库保存猴子信息：

```
session.save(monkey);
```

编程人员如果要向数据库保存一个 Monkey 对象，无须了解到底要把它保存到数据库的哪张表中，也无须了解这张表的结构，他只需调用 Hibernate API 中 Session 接口的 save() 方法就行，至于向数据库中保存 Monkey 对象的实现细节，就由 Hibernate 来代劳了。

本章范例程序位于配套光盘的 sourcecode/chapter1 目录下。运行该程序的步骤如下。

(1) 启动 MySQL 服务器，在 MySQL 服务器中创建 SAMPLEDB 数据库，然后在该数据库中创建 MONKEYS 表，相关的 SQL 脚本文件为 schema/sampledbsql。确保 MySQL 服务器具有 root 账号，并且口令为“1234”，因为本范例通过这个账号连接到数据库。

(2) 在 DOS 命令行下进入 chapter1 根目录，然后输入命令：

```
ant rungui
```

以上命令将利用 ANT 工具来编译所有的 Java 源代码，然后运行 MonkeyGui 类的 main() 方法。

如果读者不熟悉 ANT 工具或者 MySQL 的用法，可以参考本书第 2 章的 2.6.1 节（创建运行本书范例的系统环境）。

## 第2章 第一个Hibernate应用

通过学习第1章，大家已经跟随悟空对 Hibernate 有了初步了解。Hibernate 是 Java 应用和关系数据库之间的桥梁，它能进行 Java 对象和关系数据之间的映射。Hibernate 内部封装了通过 JDBC API 访问数据库的操作，向上层应用提供了更加面向对象的数据库访问 API。

在 Java 应用中使用 Hibernate 包含以下步骤。

- (1) 创建 Hibernate 的配置文件。
- (2) 创建持久化类。所谓持久化类，就是其实例需要保存到数据库中的类。
- (3) 创建对象-关系映射文件。
- (4) 通过 Hibernate API 编写访问数据库的代码。

在本章，悟空将创建一个简单的 monkeys 应用。这个例子演示如何运用 Hibernate 来访问关系数据库。monkeys 应用的功能非常简单：通过 Hibernate 来保存、更新、删除、加载及查询 Monkey 对象。图 2-1 显示了 Hibernate 在 monkeys 应用中所处的位置。

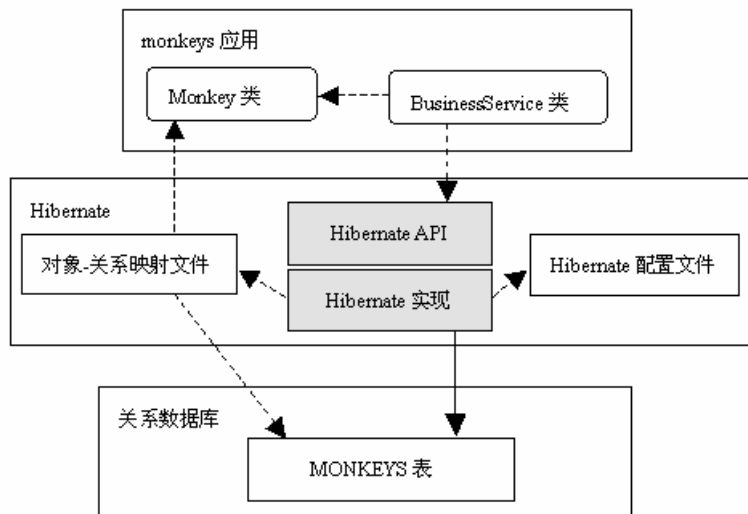


图 2-1 Hibernate 在 monkeys 应用中所处的位置

### 2.1 创建Hibernate的配置文件

Hibernate 从其配置文件中读取和数据库连接有关的信息，这个配置文件应该位

于应用的 classpath 中。Hibernate 的配置文件有两种形式：一种是 XML 格式的文件；还有一种是 Java 属性文件，采用“键=值”的形式。

本书第 3 章的 3.3 节会介绍 XML 格式的配置文件的创建方法。下面介绍如何以 Java 属性文件的格式来创建 Hibernate 的配置文件。这种配置文件的默认文件名为 hibernate.properties，例程 2-1 为示范代码。

例程 2-1 hibernate.properties

```
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost:3306/SAMPLEDB
hibernate.connection.username=root
hibernate.connection.password=1234
hibernate.show_sql=true
```

以上 hibernate.properties 文件包含了一系列属性及其属性值，Hibernate 将根据这些属性来连接数据库，本例为连接 MySQL 数据库的配置代码。表 2-1 对以上 hibernate.properties 文件中的所有属性做了描述。

表 2-1 Hibernate 配置文件的属性

属 性	描 述
hibernate.dialect	指定数据库使用的 SQL 方言
hibernate.connection.driver_class	指定数据库的驱动程序
hibernate.connection.url	指定连接数据库的 URL
hibernate.connection.username	指定连接数据库的用户名
hibernate.connection.password	指定连接数据库的口令
hibernate.show_sql	如果为 true，表示在程序运行时，会在控制台输出 SQL 语句，这有利于跟踪 Hibernate 的运行状态，默认为 false。在应用开发和测试阶段，可以把这个属性设为 true，以便跟踪和调试应用程序；在应用发布阶段，应该把这个属性设为 false，以便减少应用的输出信息，提高运行性能

Hibernate 能够访问多种关系数据库，如 MySQL、Oracle 和 Sybase 等。尽管多数关系数据库都支持标准的 SQL 语言，但是它们往往还有各自的 SQL 方言，就像不同地区的人既能说标准的普通话，还能讲各自的方言一样。hibernate.dialect 属性用于指定被访问数据库所使用的 SQL 方言，当 Hibernate 生成 SQL 查询语句时会参考本地数据库的 SQL 方言。

## 2.2 创建持久化类

持久化类是指其实例需要被 Hibernate 持久化到数据库中的类。例程 2-2 定义了一个名为 Monkey 的持久化类。



例程 2-2 Monkey.java

```
package mypack;

public class Monkey{
    private Long id;
    private String name;
    private int age;
    private char gender;

    public Monkey(){}

    public Long getId(){
        return id;
    }

    private void setId(Long id){
        this.id = id;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name=name;
    }

    public int getAge(){
        return age;
    }

    public void setAge(int age){
        this.age =age ;
    }

    public char getGender(){
        return gender;
    }

    public void setGender(char gender){
        this.gender =gender ;
    }
}
```

持久化类符合 JavaBean 的规范，包含一些属性，以及与之对应的 getXXX()和 setXXX()方法，这些 getXXX()和 setXXX()方法可以采用任意的访问级别，包括：

public、protected、默认和 private。getXXX()和 setXXX()方法必须符合特定的命名规则，“get”和“set”后面紧跟属性的名字，并且属性名的首字母为大写，如 name 属性的 get 方法为 getName()，如果把 get 方法命名为 getname()或者 getNAME()，会导致 Hibernate 在运行时抛出以下异常：

```
[java] org.hibernate.PropertyNotFoundException:
      Could not find a getter for name in class mypack.Monkey
```

在默认情况下，当 Hibernate 试图读取一个 Monkey 对象的 name 属性时，会根据 JavaBean 的命名规则，自动去调用 Monkey 对象的 getName()方法，如果不存在这个方法，就会抛出以上异常。

Monkey 持久化类有一个 id 属性，用来唯一标识 Monkey 类的每个对象。在面向对象术语中，这个 id 属性被称为对象标识符（OID，Object Identifier），通常它都用整数表示，当然也可以设为其他类型。如果 monkeyA.getId().equals(monkeyB.getId())的结果是 true，就表示 monkeyA 和 monkeyB 对象指的是同一个猴子，它们和 MONKEYS 表中的同一条记录对应。

Hibernate 要求持久化类必须提供一个不带参数的默认构造方法，因为在程序运行时，Hibernate 会运用 Java 反射机制，调用 java.lang.reflect.Constructor.newInstance()方法来构造持久化类的实例。

## 2.3 创建数据库 Schema

在本例中，与 Monkey 类对应的数据库表名为 MONKEYS，它在 MySQL 数据库中的 DDL 定义如下：

```
create table MONKEYS (
    ID bigint not null auto_increment primary key,
    NAME varchar(15) not null,
    AGE int ,
    GENDER char(1) );
```

MONKEYS 表有一个 ID 字段，它是表的代理主键，它和 Monkey 类的 id 属性对应。

## 2.4 创建对象-关系映射文件

Hibernate 采用 XML 格式的文件来指定对象和关系数据之间的映射关系。在运行时，Hibernate 将根据这个映射文件来生成各种 SQL 语句。在本例中，将创建一个名为 Monkey.hbm.xml 的文件，它用于把 Monkey 类映射到 MONKEYS 表，这个

文件应该和 `Monkey.class` 文件存放在同一个目录下。例程 2-3 为 `Monkey.hbm.xml` 文件的代码。

例程 2-3 `Monkey.hbm.xml`

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="mypack.Monkey" table="MONKEYS">

    <id name="id" column="ID" type="long">
      <generator class="increment"/>
    </id>
    <property name="name" column="NAME" type="string" not-null="true" />
    <property name="age" column="AGE" type="int" />
    <property name="gender" column="GENDER" type="character" />
  </class>

</hibernate-mapping>
```

例程 2-3 的 `Monkey.hbm.xml` 文件用于映射 `Monkey` 类。如果需要映射多个持久化类，那么既可以在同一个映射文件中映射所有类；也可以为每个类创建单独的映射文件，映射文件和类同名，扩展名为“`hbm.xml`”。后一种做法更值得推荐，因为在团队开发中，这有利于管理和维护映射文件。

`<class>` 元素指定类和表的映射，它的 `name` 属性设定类名，`table` 属性设定表名。以下代码表明和 `Monkey` 类对应的表为 `MONKEYS` 表：

```
<class name="mypack.Monkey" table="MONKEYS">
```

`<class>` 元素包含一个 `<id>` 子元素及多个 `<property>` 子元素。`<id>` 子元素设定持久化类的 `OID` 和表的主键的映射。以下代码表明 `Monkey` 类的 `id` 属性和 `MONKEYS` 表中的 `ID` 字段对应。

```
<id name="id" column="ID" type="long">
  <generator class="increment"/>
</id>
```

`<id>` 元素的 `<generator>` 子元素指定对象标识符生成器，它负责为对象的 `OID` 生成唯一标识符。本书第 4 章（映射对象标识符）将详细介绍 Hibernate 提供的各种对象标识符生成器的用法。

`<property>` 子元素设定类的属性和表的字段的映射。`<property>` 子元素主要包括 `name`、`type`、`column` 和 `not-null` 属性。

### 1. <property>元素的name属性

<property>元素的 name 属性指定持久化类的属性的名字。

### 2. <property>元素的type属性

<property>元素的 type 属性指定 Hibernate 映射类型。Hibernate 映射类型是 Java 类型与 SQL 类型的桥梁。表 2-2 列出了 Monkey 类的属性的 Java 类型、Hibernate 映射类型，以及 MONKEYS 表的字段的 SQL 类型这三者之间的对应关系。

表 2-2 Java 类型、Hibernate 映射类型及 SQL 类型之间的对应关系

Monkey 类的属性	Java 类型	Hibernate 映射类型	MONKEYS 表的字段	SQL 类型
name	java.lang.String	string	NAME	VARCHAR(15)
age	int	int	AGE	INT
gender	char	character	GENDER	CHAR(1)

如果没有为某个属性显式设定映射类型，Hibernate 会运用 Java 反射机制先识别出持久化类的特定属性的 Java 类型，然后自动使用与之对应的默认的 Hibernate 映射类型。例如，Monkey 类的 name 属性为 java.lang.String 类型，与 java.lang.String 对应的默认的映射类型为 string，因此以下两种设置方式是等价的：

```
<property name="name" column="NAME" />
```

或者：

```
<property name="name" column="NAME" type="string" />
```

### 3. <property>元素的not-null属性

如果<property>元素的 not-null 属性为 true，表明不允许为 null，默认为 false。例如以下代码表明不允许 Monkey 类的 name 属性为 null：

```
<property name="name" column="NAME" type="string" not-null="true" />
```

Hibernate 在持久化一个 Monkey 对象时，会先检查它的 name 属性是否为 null，如果为 null，就会抛出以下异常：

```
[java] Exception in thread "main" org.hibernate.PropertyValueException:
not-null property references a null or transient value: mypack.Monkey.name
```

### 4. <property>元素的column属性

<property>元素的 column 属性指定与类的属性映射的表的字段名。以下代码表明和 age 属性对应的字段为 AGE 字段：

```
<property name="age" column="AGE" type="int" />
```

如果没有设置<property>元素的 column 属性，Hibernate 将直接以类的属性名作为字段名，也就是说，在默认情况下，Hibernate 认为与 Monkey 类的 age 属性对应的字段为 AGE 字段。

<property>元素还可以包括<column>子元素，它和<property>元素的 column 属性一样，都可以设定与类的属性映射的表的字段名。以下两种设置方式是等价的：

```
<property name="age" column="AGE" type="int"/>
```

或者：

```
<property name="age" type="int">
  <column name="AGE" />
</property>
```

图 2-2 显示了 Monkey.hbm.xml 配置的对象-关系映射。

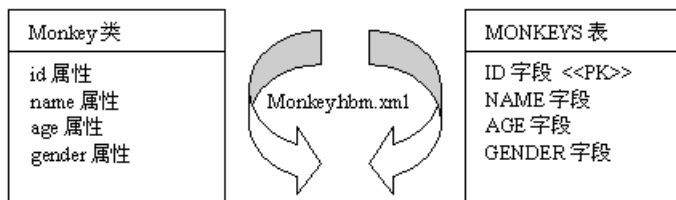


图 2-2 Monkey.hbm.xml 配置的对象-关系映射

## 2.5 通过Hibernate API操纵数据库

Hibernate 对 JDBC API 进行了封装，提供了更加面向对象的 API。以下图 2-3 和图 2-4 对比了直接通过 JDBC API 及通过 Hibernate API 来访问数据库的两种方式。

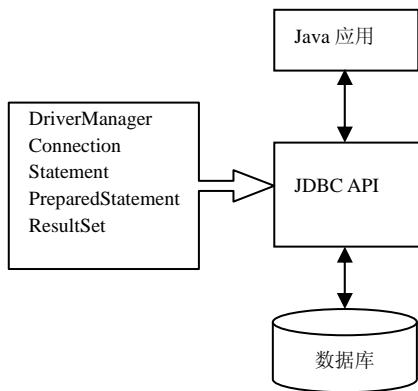


图 2-3 通过 JDBC API 访问数据库

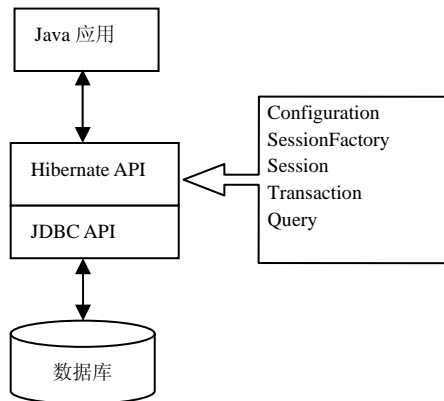


图 2-4 通过 Hibernate API 访问数据库

Hibernate API 中主要包括以下接口：

- **Configuration 接口**：用于配置并且启动 Hibernate。Hibernate 应用通过 Configuration 实例来获得对象-关系映射文件中的元数据，以及动态配置 Hibernate 的属性，然后创建 SessionFactory 实例。

- **SessionFactory 接口**：一个 SessionFactory 实例对应一个数据存储源，应用从 SessionFactory 中获得 Session 实例。
- **Session 接口**：是 Hibernate 应用使用最广泛的接口。Session 也被称为持久化管理器，它提供了和持久化相关的操作，如保存、更新、删除和加载对象。
- **Transaction 接口**：是 Hibernate 的数据库事务接口，它对底层的事务接口做了封装，底层事务接口包括 JDBC 事务和 JTA (Java Transaction API) 事务。
- **Query 接口**：是 Hibernate 的查询接口，用于向数据库查询对象，以及控制执行查询的过程。Query 实例包装了一个 HQL (Hibernate Query Language) 查询语句，HQL 查询语句与 SQL 查询语句有些相似，但 HQL 查询语句是面向对象的，它引用类名及类的属性名，而不是表名及表的字段名。

以下例程 2-4 的 BusinessService 类演示了通过 Hibernate API 对 Monkey 对象进行持久化的操作。

例程 2-4 BusinessService.java

```
package mypack;

import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class BusinessService{
    public static SessionFactory sessionFactory;

    /** 初始化 Hibernate, 创建 SessionFactory 实例 */
    static{
        try{
            // 根据默认位置的 Hibernate 配置文件的配置信息, 创建一个 Configuration 实例
            Configuration config = new Configuration();
            // 加载 Monkey 类的对象-关系映射文件
            config.addClass(Monkey.class);
            // 创建 SessionFactory 实例
            sessionFactory = config.buildSessionFactory();
        }catch(RuntimeException e){e.printStackTrace();throw e;}
    }

    /** 查询所有的 Monkey 对象, 然后打印 Monkey 对象信息 */
    public void findAllMonkeys(){
        Session session = sessionFactory.openSession(); // 创建一个会话
        Transaction tx = null;
        try {
            tx = session.beginTransaction(); // 开始一个事务
            Query query=session.createQuery("from Monkey as m order by m.name asc");
```

```
List monkeys=query.list();
for (Iterator it = monkeys.iterator(); it.hasNext();) {
    Monkey monkey=(Monkey) it.next();
    System.out.println("ID="+monkey.getId()
        +",姓名="+monkey.getName()
        +",年龄="+monkey.getAge()
        +",性别="+((monkey.getGender()=="M"? "公猴": "母猴"));
}

tx.commit(); //提交事务

} catch (RuntimeException e) {
    if (tx != null) {
        tx.rollback();
    }
    throw e;
} finally {
    session.close();
}
}

/** 持久化一个 Monkey 对象 */
public void saveMonkey(Monkey monkey){
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.save(monkey);
        tx.commit();

    } catch (RuntimeException e) {
        if (tx != null) {
            tx.rollback();
        }
        throw e;
    } finally {
        session.close();
    }
}

/** 按照 OID 加载一个 Monkey 对象, 然后修改它的属性 */
public void loadAndUpdateMonkey(Long monkey_id,int age){.....}

/**删除 Monkey 对象 */
public void deleteMonkey(Monkey monkey){.....}
```



```

public void test(){
    Monkey monkey=new Monkey();
    monkey.setName("智多星");
    monkey.setAge(1);
    monkey.setGender('M');

    saveMonkey(monkey);

    findAllMonkeys();
    loadAndUpdateMonkey(monkey.getId(),2);
    findAllMonkeys();
    deleteMonkey(monkey);
}

public static void main(String args[]){
    new BusinessService().test();
    sessionFactory.close();
}
}

```

以上例子演示了通过 **Hibernate API** 访问数据库的一般流程。首先应该在应用的启动阶段对 **Hibernate** 进行初始化，然后就可以通过 **Hibernate** 的 **Session** 接口来访问数据库。

### 2.5.1 Hibernate的初始化

**BusinessService** 类的静态代码块负责 **Hibernate** 的初始化工作，如读取 **Hibernate** 的配置信息及对象-关系映射信息，最后创建 **SessionFactory** 实例。当 **JVM**（Java 虚拟机）加载 **BusinessService** 类时，会执行该静态代码块。初始化过程包括如下步骤。

（1）创建一个 **Configuration** 类的实例，**Configuration** 类的构造方法把默认文件路径下的 **hibernate.properties** 配置文件中的配置信息读入到内存：

```
Configuration config = new Configuration();
```

（2）调用 **Configuration** 类的 **addClass(Monkey.class)**方法：

```
config.addClass(Monkey.class);
```

该方法把默认文件路径下的 **Monkey.hbm.xml** 文件中的映射信息读入到内存中。

（3）调用 **Configuration** 类的 **buildSessionFactory()**方法：

```
sessionFactory = config.buildSessionFactory();
```

该方法创建一个 **SessionFactory** 实例，并把 **Configuration** 对象包含的所有配置信息复制到 **SessionFactory** 对象的缓存中。**SessionFactory** 代表一个数据库存储源，如

果应用只有一个数据库存储源，那么只需创建一个 `SessionFactory` 实例。当 `SessionFactory` 对象创建后，该对象不和 `Configuration` 对象关联。因此，如果再修改 `Configuration` 对象包含的配置信息，不会对 `SessionFactory` 对象的行为有任何影响。

创建一个 `SessionFactory` 对象，需要消耗很多资源，因此它是一个重量级对象。如果应用只有一个数据存储源，只需创建一个 `SessionFactory` 实例，因为随意地创建 `SessionFactory` 实例会占用大量内存空间。

Hibernate 的许多类和接口都支持方法链编程风格，`Configuration` 类的 `addClass()` 方法返回当前 `Configuration` 实例，因此对于以下代码：

```
Configuration config = new Configuration();
config.addClass(Monkey.class);
sessionFactory = config.buildSessionFactory();
```

如果使用方法链编程风格，可以改写为：

```
sessionFactory = new Configuration()
    .addClass(Monkey.class)
    .buildSessionFactory();
```

方法链编程风格能使应用程序代码更加简捷。在使用这种编程风格时，最好把每个调用方法放在不同的行，否则在跟踪程序时，无法跳入每个调用方法中。

### 2.5.2 访问Hibernate的Session接口

初始化过程结束后，就可以调用 `SessionFactory` 实例的 `openSession()` 方法来获得 `Session` 实例，然后通过它执行访问数据库的操作。`Session` 接口提供了操纵数据库的各种方法，如：

- `save()` 方法：把对象保存到数据库中。
- `update()` 方法：更新数据库中的对象。
- `delete()` 方法：把特定的对象从数据库中删除。
- `get()` 方法或 `load()` 方法：从数据库中加载对象。

`Session` 是一个轻量级对象。通常将每一个 `Session` 实例和一个数据库事务绑定，也就是说，每执行一个数据库事务，都应该先创建一个新的 `Session` 实例。如果事务执行中出现异常，应该撤销事务。不论事务执行成功与否，最后都应该调用 `Session` 的 `close()` 方法，从而释放 `Session` 实例占用的资源。以下代码演示了用 `Session` 来执行事务的流程，其中 `Transaction` 类用来控制事务。

```
Session session = factory.openSession();
Transaction tx;
try {
    //开始一个事务
```

```

    tx = session.beginTransaction();
    //执行事务
    ...
    //提交事务
    tx.commit();
}
catch (RuntimeException e) {
    //如果出现异常，就撤销事务
    if (tx!=null) tx.rollback();
    throw e;
}finally {
    //不管事务执行成功与否，最后都关闭 Session
    session.close();
}

```

图 2-5 为正常执行数据库事务（即没有发生异常）的时序图。

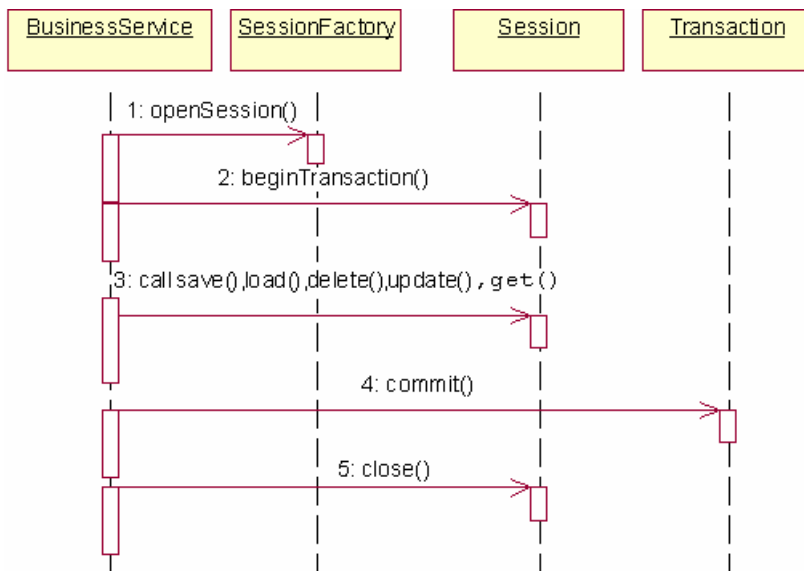


图 2-5 正常执行数据库事务的时序图

BusinessService 类提供了保存、删除、查询和更新 Monkey 对象的各种方法。BusinessService 类的主方法调用 test()方法，test()方法又调用以下方法：

### 1. saveMonkey()方法

该方法调用 Session 的 save()方法，把 Monkey 对象持久化到数据库中。

```

tx = session.beginTransaction();
session.save(monkey);
tx.commit();

```

当运行 session.save()方法时，Hibernate 执行以下 SQL 语句：

```
insert into MONKEYS (ID, NAME, AGE, GENDER)
values(1,'智多星','1','M')
```

## 2. findAllMonkeys()方法

该方法通过 Query 接口来查询所有的 Monkey 对象。

```
tx = session.beginTransaction(); //开始一个事务
Query query=session.createQuery("from Monkey as m order by m.name asc");
List monkeys=query.list();
for (Iterator it = monkeys.iterator(); it.hasNext();) {
    Monkey monkey=(Monkey) it.next();
    System.out.println("ID="+monkey.getId()
        +",姓名="+monkey.getName()
        +",年龄="+monkey.getAge()
        +",性别="+ (monkey.getGender()=='M'? "公猴": "母猴"));
}

tx.commit(); //提交事务
```

Query 接口是 Hibernate API 中的查询接口。以上代码先通过 Session 的 createQuery()方法创建了一个 Query 实例。以上代码向 Session 的 createQuery()方法传递的参数为“from Monkey as m order by m.name asc”，它使用的是面向对象的 HQL (Hibernate Query Language, Hibernate 查询语言)，本书第 7 章 (Hibernate 的检索策略和检索方式) 将介绍这种查询语言。运行 query.list()方法时，Hibernate 执行以下 SQL 语句：

```
select * from MONKEYS order by NAME asc;
```

## 3. loadAndUpdateMonkey ()方法

该方法调用 Session 的 get()方法，加载 Monkey 对象，然后再修改 Monkey 对象的属性。

```
tx = session.beginTransaction();
Monkey m=(Monkey)session.get(Monkey.class,monkey_id);
m.setAge(age);
tx.commit();
```

以上代码先调用 Session 的 get()方法，它按照参数指定的 OID 从数据库中检索出匹配的 Monkey 对象，Hibernate 会执行以下 SQL 语句：

```
select * from MONKEYS where ID=1;
```

loadAndUpdateMonkey()方法接着修改 Monkey 对象的 age 属性。那么，Hibernate 会不会同步更新数据库中相应的 MONKEYS 表的记录呢？答案是肯定的。Hibernate 会自动进行脏检查 (dirty check)，按照内存中的 Monkey 对象的状态的变化，来同步更新数据库中相关的数据，Hibernate 会执行以下 SQL 语句：

```
update MONKEYS set NAME='智多星',AGE=2,GENDER='M' where ID=1;
```

尽管只有 Monkey 对象的 age 属性发生了变化,但是 Hibernate 执行的 update 语句中会包含所有的字段。

## 5. deleteMonkey()方法

该方法调用 Session 的 delete()方法,删除参数指定的 Monkey 对象:

```
tx = session.beginTransaction();
session.delete(monkey);
tx.commit();
```

运行 session.delete()方法时,Hibernate 根据 Monkey 对象的 OID,执行以下 delete 语句:

```
delete from MONKEYS where ID=1;
```

## 2.6 运行monkeys应用

monkeys 应用虽然简单,却是麻雀虽小,五脏俱全。要让 monkeys 应用运行起来,需要调兵遣将,动用多个软件,如 JDK、ANT、Hibernate 和 MySQL。本节将介绍安装这些软件的方法、monkeys 应用的目录结构,以及运行 monkeys 应用的方法。

### 2.6.1 创建运行本书范例的系统环境

运行本书的例子,需要安装以下软件。

(1) JDK1.4、JDK1.5 或者以上版本: Hibernate 要求 JDK1.4 以上的版本,如果安装 JDK1.3,可能会导致程序无法正常编译或运行。

(2) ANT: 它是发布、编译和运行 Java 应用的工具软件。

(3) Hibernate: 它是本书介绍的对象-关系映射工具软件。

(4) MySQL: 它是本书范例使用的关系数据库。

表 2-3 列出了以上软件的下载网址。此外,本书配套光盘的 software 目录下也提供了以上软件(不包括 JDK 软件)。

表 2-3 本书使用的软件的下载网址

软 件	下载网地址
JDK1.5	java.sun.com
ANT	www.apache.org
MySQL	www.mysql.com
Hibernate	www.hibernate.org

## 1. 安装和配置软件

以上软件的安装都很简单，其中 JDK 和 MySQL 的安装软件是可运行程序，只需直接运行安装程序。对于 MySQL5.x 版本，在安装 MySQL 的过程中，会出现为“root”用户输入口令的窗口，在该窗口中设置口令为“1234”，如图 2-6 所示。在本书中，登入 MySQL 服务器的用户名一律为“root”，口令为“1234”。



图 2-6 设置 MySQL 的“root”用户的口令

此外，为了使 MySQL 能支持包括中文在内的多国语言，建议将 MySQL 使用的默认字符编码设为“UTF8”，参见图 2-7。

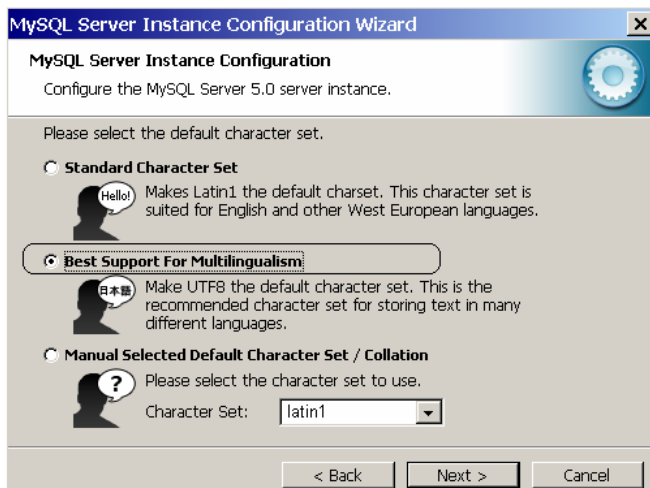


图 2-7 把 MySQL 默认的字符编码设为 UTF8

ANT 和 Hibernate 的安装软件是压缩软件包，只需把压缩文件解压到本地硬盘。安装好以上软件后需要在操作系统中设置以下系统环境变量：

- JAVA\_HOME: JDK 的安装目录。
- ANT\_HOME: ANT 的安装目录。
- PATH: 把%JAVA\_HOME%/bin 目录添加到 PATH 变量中, 以便于在当前路径下, 从 DOS 命令行直接运行 JDK 的 javac 和 java 命令; 把 %ANT\_HOME%/bin 目录添加到 PATH 变量中, 以便于在当前路径下, 从 DOS 命令行直接运行 ANT。

在 Windows XP 操作系统中创建 JAVA\_HOME 环境变量的步骤如下。

(1) 打开“控制面板”, 选择“系统”图标。

(2) 双击“系统”图标, 运行 Windows XP 系统程序, 选择“高级”标签, 如图 2-8 所示。

(3) 在图 2-8 中单击【环境变量】按钮, 将会出现设置环境变量的窗口, 如图 2-9 所示。

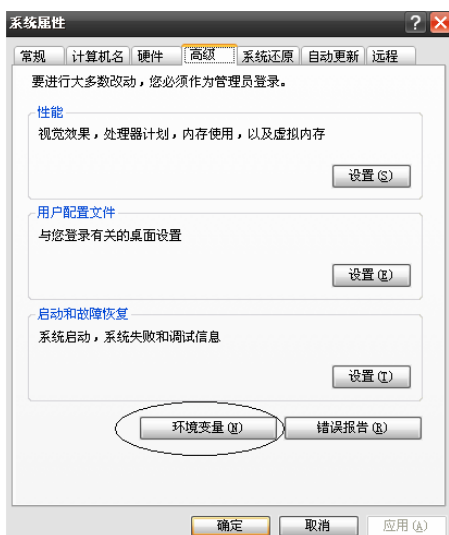


图 2-8 Windows XP 的系统程序



图 2-9 “环境变量”对话框

(4) 单击图 2-9 中“系统变量”区域的【新建】按钮, 将会出现“新建系统变量”对话框, 在对话框中新建 JAVA\_HOME 环境变量, 环境变量值为 C:\jdk1.5, 如图 2-10 所示。



图 2-10 设置 JAVA\_HOME 环境变量

接下来可以重复以上步骤创建 ANT\_HOME 环境变量。环境变量创建好以后，就可以启动各个服务器，然后运行本章范例程序。

## 2. 启动MySQL服务器

MySQL 服务器既可以作为前台服务程序运行，也可以作为后台服务程序运行。在 MySQL 安装目录的 bin 目录下提供了以下 MySQL 服务器程序：

- **mysqld.exe**：最基本的 MySQL 服务器程序。
- **mysqld-nt.exe**：Windows NT/2000/XP 平台的优化版本，支持命名管道。

执行以上任意一个程序，都会启动 MySQL 服务，它以前台服务的形式运行。此外，也可以按以下步骤把 MySQL 作为后台服务运行：

- (1) 在 DOS 下转到 MySQL 的安装目录的 bin 子目录下。
- (2) 在 Windows XP 中注册 MySQL 服务，输入命令：`mysqld-nt --install`。
- (3) 启动 MySQL 服务，输入命令：`net start mysql`。

此外，也可以通过 Windows XP 的【控制面板】→【管理工具】→【服务】程序来管理注册过的 MySQL 服务。

## 3. 运行MySQL的客户程序

(1) 在 DOS 命令行下运行 `mysql.exe` 程序。步骤为先转到 MySQL 安装目录的 bin 目录下，输入如下命令：

```
C:\mysql\bin> mysql -u root -p
```

接下来会提示输入 root 用户的口令，此处输入口令“1234”：

```
Enter password: ****
```

然后就会进入如图 2-11 所示的命令行客户程序。

(2) 创建数据库 SAMPLEDB，SQL 命令如下：

```
create database SAMPLEDB;
```

(3) 进入 SAMPLEDB 数据库，SQL 命令如下：

```
use SAMPLEDB;
```

(4) 在 SAMPLEDB 数据库中创建 MONKEYS 表，SQL 命令如下：

```
create table MONKEYS (  
    ID bigint not null auto_increment primary key,  
    NAME varchar(15) not null,  
    AGE int ,  
    GENDER char(1) ,  
);
```

(5) 退出 MySQL 客户程序，输入命令：`exit`。



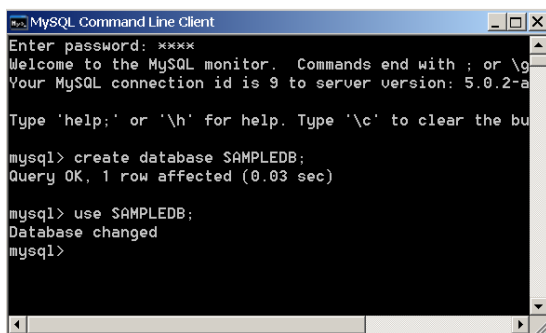


图 2-11 MySQL 的客户程序

在配套光盘的 `sourcecode\chapter2\monkeys\schema` 目录下提供了创建 SAMPLEDB 数据库和 MONKEYS 表的 SQL 脚本文件，文件名为 `sampledb.sql`。如果不想在 MySQL.exe 程序中手工输入 SQL 语句，也可以直接运行 `sampledb.sql`，步骤为先转到 MySQL 安装目录的 `bin` 目录下，输入如下命令：

```
C:\mysql\bin> mysql -u root -p <C:\monkeys\schema\sampledb.sql
```

接下来会提示输入 root 用户的口令，此处输入口令“1234”：

```
Enter password: ****
```

接下来 MySQL 客户程序就会自动执行 `C:\monkeys\schema\sampledb.sql` 文件中的所有 SQL 语句。在以上 `mysql` 命令中，“<”后面设定 SQL 脚本文件的路径。

对于本书中的所有范例，如果没有做特别说明，在运行程序前都需创建相关的数据库表，在每章范例的 `schema` 目录下都提供了相应的 SQL 脚本文件。

## 2.6.2 创建monkeys应用的目录结构

monkeys 应用的初始目录结构如图 2-12 所示。

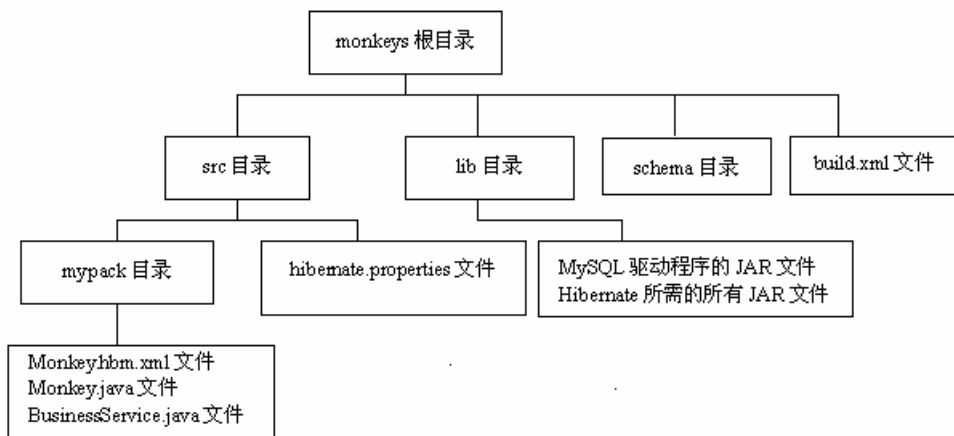


图 2-12 monkeys 应用的初始目录结构

src 子目录用于存放 Java 源文件、hibernate.properties 配置文件及 Monkey.hbm.xml 文件。lib 目录下包含了 MySQL 的驱动程序类库 mysql.driver.jar 文件，以及 Hibernate 的类库文件。

### 2.6.3 运行monkeys应用

接下来将通过 ANT 工具创建 classes 子目录，并且把 src 子目录下的 hibernate.properties 和 Monkey.hbm.xml 文件复制到 classes 目录下，此外，编译生成的 Java 类文件也放在 classes 目录的相应子目录下。

在 monkeys 应用的根目录下有一个 build.xml 文件，它是 ANT 的工程文件，参见例程 2-5。

例程 2-5 build.xml

```
<?xml version="1.0"?>
<project name="Learning Hibernate" default="prepare" basedir=".">

  <!-- Set up properties containing important project directories -->
  <property name="source.root" value="src"/>
  <property name="class.root" value="classes"/>
  <property name="lib.dir" value="lib"/>

  <!-- Set up the class path for compilation and execution -->
  <path id="project.class.path">
    <!-- Include our own classes, of course -->
    <pathelement location="${class.root}" />
    <!-- Include jars in the project library directory -->
    <fileset dir="${lib.dir}">
      <include name="*.jar"/>
    </fileset>
  </path>

  <!-- Create our runtime subdirectories and copy resources into them -->
  <target name="prepare" description="Sets up build structures">
    <delete dir="${class.root}" />
    <mkdir dir="${class.root}" />

    <!-- Copy our property files and O/R mappings for use at runtime -->
    <copy todir="${class.root}">
      <fileset dir="${source.root}">
        <include name="**/*.properties"/>
        <include name="**/*.hbm.xml"/>
        <include name="**/*.xml"/>
      </fileset>
    </copy>
  </target>
</project>
```

```

</target>

<!-- Compile the java source of the project -->
<target name="compile" depends="prepare"
        description="Compiles all Java classes">
    <javac srcdir="${source.root}"
           destdir="${class.root}"
           debug="on"
           optimize="off"
           deprecation="on">
        <classpath refid="project.class.path"/>
    </javac>
</target>

<target name="run" description="Run a Hibernate sample"
        depends="compile">
    <java classname="mypack.BusinessService" fork="true">
        <classpath refid="project.class.path"/>
    </java>
</target>

</project>

```

在 build.xml 文件中先定义了 3 个属性：

```

<property name="source.root" value="src"/>
<property name="class.root" value="classes"/>
<property name="lib.dir" value="lib"/>

```

source.root 属性指定 Java 源文件的路径，class.root 属性指定 Java 类的路径，lib.dir 属性指定所有 JAR 文件的路径。

在 build.xml 文件中接着定义了 3 个 target（任务）。

- **prepare target:** 如果存在 classes 子目录，先将它删除。接着重新创建 classes 子目录。然后把 src 子目录下所有扩展名为 “.properties”、“.hbm.xml” 和 “.xml” 文件复制到 classes 目录下。
- **compile target:** 编译 src 子目录下的所有 Java 源文件。编译生成的类文件存放在 classes 子目录下。
- **run target:** 运行 BusinessService 类。

以上 3 个 target 的依赖关系参见图 2-13。所谓依赖，是指在执行当前 target 之前必须先执行所依赖的 target。<target>元素的 depends 属性指定所依赖的 target。根据图 2-13 可以看出，当运行 run target 时，会依次执行 prepare target、compile target 和 run target。

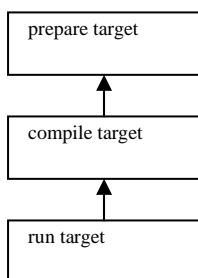


图 2-13 build.xml 文件中 3 个 target 的依赖关系

把 monkeys 应用作为独立应用程序运行的步骤如下。

(1) 启动 MySQL 服务器。

(2) 在 MySQL 服务器中安装 monkeys 应用的数据库。创建数据库的脚本为 schema/sampled.sql。在 MySQL 中运行该脚本，它负责创建 SAMPLEDB 数据库，然后在该数据库中创建 MONKEYS 表。

(3) 在 DOS 命令行下进入 monkeys 根目录，然后输入如下命令：

```
ant run
```

以上命令将依次执行 prepare target、compile target 和 run target，run target 运行 BusinessService 类的主方法，在 DOS 控制台输出很多信息，以下是部分内容：

```
[java] Hibernate: select max(ID) from MONKEYS
[java] Hibernate: insert into MONKEYS (NAME, AGE, GENDER, ID)
values (?, ?, ?, ?)
[java] Hibernate: select monkey0_.ID as ID0_, monkey0_.NAME as NAME0_,
monkey0_.AGE as AGE0_, monkey0_.GENDER as GENDER0_
from MONKEYS monkey0_ order by monkey0_.NAME asc
[java] ID=1,姓名=智多星,年龄=1,性别=公猴
[java] Hibernate: select monkey0_.ID as ID0_0_, monkey0_.NAME as NAME0_0_,
monkey0_.AGE as AGE0_0_, monkey0_.GENDER as GENDER0_0_
from MONKEYS monkey0_ where monkey0_.ID=?
[java] Hibernate: update MONKEYS set NAME=?, AGE=?, GENDER=? where ID=?
[java] Hibernate: select monkey0_.ID as ID0_, monkey0_.NAME as NAME0_,
monkey0_.AGE as AGE0_, monkey0_.GENDER as GENDER0_
from MONKEYS monkey0_ order by monkey0_.NAME asc
[java] ID=1,姓名=智多星,年龄=2,性别=公猴
[java] Hibernate: delete from MONKEYS where ID=?
```

由于 hibernate.properties 文件的 show\_sql 属性为 true，因此，Hibernate 把运行时执行的 SQL 语句输出到了控制台。例如，当运行 session.get(Monkey.class, monkey\_id)方法时，Hibernate 执行的 SQL 语句为：

```
select monkey0_.ID as ID0_0_, monkey0_.NAME as NAME0_0_,
monkey0_.AGE as AGE0_0_, monkey0_.GENDER as GENDER0_0_
```

```
from MONKEYS monkey0_ where monkey0_.ID=?
```

图 2-14 显示了运行完 run target 之后的 monkeys 应用的主要目录结构，其中，classes 子目录及它包含的内容是新建的。

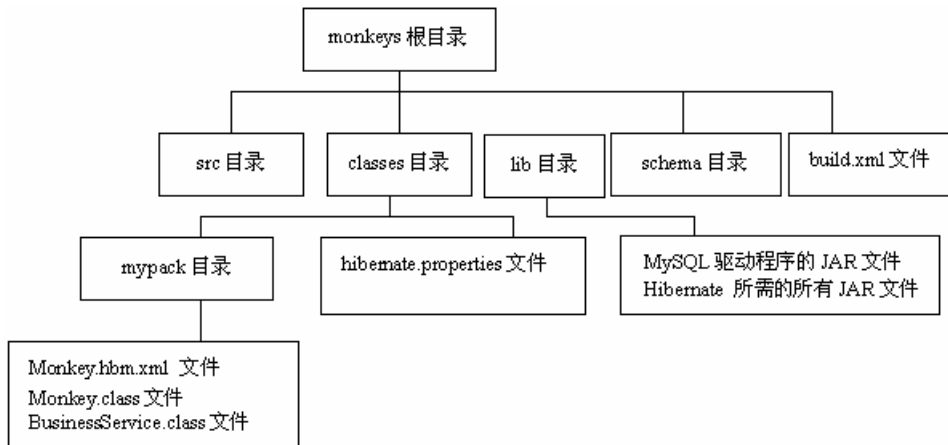


图 2-14 运行完 run target 之后的 monkeys 应用的主要目录结构

## 2.6.4 给monkeys应用加入用户界面

第 1 章已经创建了用户界面类 MonkeyGui。可以在 build.xml 文件中再加入一个 rungui target，它负责运行 MonkeyGui 类：

```
<target name="rungi" description="Run a Hibernate sample"
  depends="compile" >
  <java classname="mypack.MonkeyGui" fork="true">
    <classpath refid="project.class.path"/>
  </java>
</target>
```

在 DOS 命令行下进入 monkeys 根目录，然后输入如下命令：

```
ant rungui
```

以上命令将运行 MonkeyGui 类，显示 MonkeyGui 类创建的图形用户界面。当用户通过该界面保存一个猴子信息时，MonkeyGui 类会调用 BusinessService 类的 saveMonkey() 方法。

## 2.7 小结

在这一章，悟空借助 Hibernate，成功地把小猴智多星的信息保存到关系数据库中。对于活蹦乱跳的小猴智多星，在 Java 程序眼里，它是一个 OID 为 1 的 Monkey

对象；在数据库眼里，它是 MONKEYS 表中 ID 为 1 的一条记录。Hibernate 参考 Monkey.hbm.xml 对象-关系映射文件，就能准确地对 Monkey 对象和 MONKEYS 记录进行映射。

悟空小试身手，创建了简单的 monkeys 应用。这个应用虽然简单，却完整地展示了通过 Hibernate 来访问数据库的基本步骤：

- (1) 创建 Hibernate 的配置文件，在配置文件中提供连接特定数据库的信息。
- (2) 创建持久化类。持久化类符合 JavaBean 的规范。
- (3) 创建对象-关系映射文件，Hibernate 根据该映射文件来生成 SQL 语句。

(4) 在应用程序中通过 Hibernate API 来访问数据库。在应用启动时先初始化 Hibernate，创建一个 SessionFactory 实例；接下来每次执行数据库事务时，先从 SessionFactory 中获得一个 Session 实例，再通过 Session 实例来保存、更新、删除、加载 Java 对象，以及通过 Query 实例来查询 Java 对象。

## 第3章 对象-关系映射基础

在第2章，悟空已经掌握了对象-关系映射的基本方法，能够顺利地把 Monkey 类映射到 MONKEYS 表。如下表 3-1 所示，对象模型和关系数据模型之间存在以下基本映射关系。

表 3-1 对象-关系的基本映射

对象模型	关系数据模型
类	表
对象	表的行（即记录）
属性	表的列（即字段）

以 Monkey 类与 MONKEYS 表为例，一个 Monkey 对象与 MONKEYS 表中的一条记录对应。Monkey 类的 id 属性与 MONKEYS 表的 ID 字段对应；Monkey 类的 name 属性与 MONKEYS 表的 NAME 字段对应，以此类推。

但是，假如 Monkey 类中有 firstname 属性和 lastname 属性，但没有 name 属性，而在数据库中的 MONKEYS 表中只有 NAME 字段，在这种情况下该如何映射呢？

在本章，大家将跟随悟空进一步了解单个持久化类与单个数据库表之间进行映射的技巧。本章主要解决以下映射问题：

- 持久化类的属性没有相关的 getXXX()和 setXXX()方法。
- 持久化类的属性在数据库表中没有对应的字段，或者数据库表中的字段在持久化类中没有对应的属性。
- 控制 Hibernate 生成的 insert 和 update 语句。

### 3.1 持久化类的属性及访问方法

持久化类使用 JavaBean 的风格，为需要被访问的属性提供 getXXX()和 setXXX()方法，这两个方法也称为持久化类的访问方法。例如，Monkey 类有一个 name 属性，代表猴子名字，与此对应，Monkey 类提供了 getName()和 setName()方法。外部程序通过 getName()方法来读取 Monkey 对象的 name 属性，例如：

```
System.out.println(monkey.getName());
```

外部程序通过 setName()方法来修改 Monkey 对象的 name 属性，例如：

```
monkey.setName("智多星");
```

在 Hibernate 应用中，持久化类的访问方法有两个调用者：

- **Java 应用程序：**调用 Monkey 对象的 `getXXX()`方法，读取 Monkey 信息，把它输出到用户界面。调用 Monkey 对象的 `setXXX()`方法，把用户输入的 Monkey 信息写入到 Monkey 对象中。
- **Hibernate：**调用 Monkey 对象的 `getXXX()`方法，读取 Monkey 信息，把它保存到数据库。调用 Monkey 对象的 `setXXX()`方法，把从数据库中读出的 Monkey 信息写入到 Monkey 对象。

图 3-1 显示了 Java 应用程序和 Hibernate 调用 Monkey 对象的访问方法的过程。

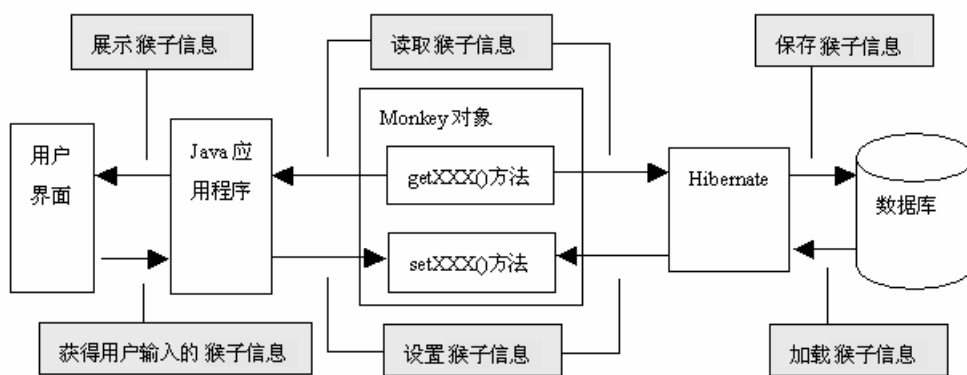


图 3-1 持久化类的访问方法的两个调用者

确切地说，当 Hibernate 的 Session 执行 `save()`、`update()` 或 `saveOrUpdate()` 方法时会调用 Monkey 对象的 `getXXX()` 方法；当 Session 执行 `get()` 或 `load()` 方法，以及 Query 执行查询操作时会调用 Monkey 对象的 `setXXX()` 方法。

值得注意的是，Java 应用程序不能访问持久化类的 `private` 类型的 `getXXX()` 和 `setXXX()` 方法，而 Hibernate 没有这个限制，它能够访问各种访问级别的 `getXXX()` 和 `setXXX()` 方法，Java 访问级别包括：`private`、默认、`protected` 和 `public`。

### 3.1.1 基本类型属性和包装类型属性

Java 有 8 种基本类型：`byte`、`short`、`char`、`int`、`long`、`float`、`double` 和 `boolean`，与此对应，Java 提供了 8 种包装类型：`Byte`、`Short`、`Character`、`Integer`、`Long`、`Float`、`Double` 和 `Boolean`。基本类型与包装类型之间可以方便地转换，例如：

```
double primitiveDouble=1;
//把 double 基本类型转换为 Double 包装类型
Double wrapperDouble= new Double(primitiveDouble);
//把 Double 包装类型转换为 double 基本类型
primitiveDouble=wrapperDouble.doubleValue();
```



在持久化类中，既可以把属性定义为基本类型，也可以定义为包装类型，它们对应相同的 Hibernate 映射类型。例如，不管 Monkey 类的 age 属性是 int 类型，还是 Integer 类型，都采用如下映射方式：

```
<property name="age" type="int" column="AGE"/>
```

基本类型和包装类型各有优缺点。基本类型的优点是使用方便，可以直接把它显示到用户界面上，而且对于数字类型，可以直接进行数学运算，例如：

```
double price1=100;
double price2=100;
double price3=price1+price2; //数学运算
```

相对而言，包装类型使用起来要稍微麻烦些，例如在 JDK1.4 及以前的版本中，对于数字类型的包装类型，必须先转换为基本类型，才能进行数学运算：

```
Double price1=new Double(100);
Double price2=new Double(100);
Double price3=new Double(price1.doubleValue()+price2.doubleValue());
```

幸运的是，从 JDK1.5 版本开始，数字类型的包装类型可以直接参与数学运算，这大大简化了包装类型的使用方式。例如在 JDK1.5 中，以下代码都是合法的：

```
double d1=11;
double d2=new Double(11)+new Double(11)+d1; //d2 的值为 33.0
Double d3=new Double(11)+new Double(11)+d1; //d3 的值为 33.0
```

基本类型的缺点在于无法表达 null 值。所有基本类型的默认值都不是 null，如数字类型的基本类型的默认值为 0。在某些场合，可以用基本类型的默认值来表示属性值是未知的。例如，对于 Monkey 类的 int 类型的 age 属性，如果 age 属性为 0，可以把它理解为该猴子的年龄是未知的。

但在某些场合，如果“0”本身包含特定业务含义，就无法用“0”来表示未知状态。例如 Student 类有一个 int 类型的 score 属性，表示学生的考试分数。int 类型的 score 属性无法表达这样的业务需求：

- 如果 score 属性为 null，表示该学生的成绩是未知的，有可能得了 100 分，也有可能得了 0 分，只是暂时还不知道成绩。
- 如果 score 属性为 0，表示学生考试成绩为 0 分。

在以上情况中，必须使用包装类型。包装类型都是 Java 类，它们的默认值是 null。如果把 Student 类的 score 属性定义为 Integer 类型，score 属性既可以取默认值 null，也可以被显式赋值为 0：

```
student.setScore(new Integer(0));
```

在 SQL 中，所有数据类型的默认值都是 null，当通过 insert 语句向 STUDENTS

表插入一条记录时，如果没有为 **SCORE** 字段赋值，那么这个字段就被自动赋值为 **null**，当然，也可以在 **insert** 语句中显式地把 **SCORE** 字段赋值为 **null**。可见，Java 包装类型与 SQL 数据类型之间具有更直接的对应关系。

Hibernate 既支持包装类型，也支持基本类型。开发人员可以根据编程习惯及业务需求来决定使用何种类型。对于持久化类的 **OID**，推荐使用包装类型，Hibernate API 对 Java 包装类型提供了友好的支持，它的接口或类的许多方法都接受包装类型的参数，例如：

```
Monkey monkey=(Monkey)session.get(Monkey.class, new Long(1));
```

### 3.1.2 Hibernate 访问持久化类属性的策略

在对象-关系映射文件中，**<property>** 元素的 **access** 属性用于指定 Hibernate 访问持久化类的属性的方式。**access** 属性有以下两个可选值。

- **property**：这是默认值，表明 Hibernate 通过相应的 **setXXX()** 和 **getXXX()** 方法来访问类的属性。这是优先推荐的方式，为持久化类的每个属性提供 **setXXX()** 和 **getXXX()** 方法，可以更灵活地封装持久化类，提高对象模型的透明性。
- **field**：表明 Hibernate 运用 Java 反射机制直接访问类的属性。例如，如果 **Monkey** 类没有为 **name** 属性提供 **setName()** 和 **getName()** 方法，就可以把 **access** 属性设为 **field**，使 Hibernate 能直接访问 **name** 属性：

```
<property name="name" access="field" column="NAME" />
```

### 3.1.3 在持久化类的访问方法中加入程序逻辑

在持久化类的访问方法中，可以加入程序逻辑，下面举例说明。

#### 1. 在 Monkey 类的 **getName()** 和 **setName()** 方法中加入程序逻辑

假定在 **Monkey** 类中有 **firstname** 属性和 **lastname** 属性，但没有 **name** 属性，而在数据库中的 **MONKEYS** 表中只有 **NAME** 字段。当 Hibernate 从数据库中取得了 **MONKEYS** 表的 **NAME** 字段值后，会调用 **setName()** 方法，此时应该让 Hibernate 通过 **setName()** 方法来自动设置 **firstname** 属性和 **lastname** 属性。这需要在 **setName()** 方法中加入额外的程序逻辑，参见例程 3-1。

例程 3-1 Monkey.java

```
package mypack;
import java.util.StringTokenizer;
public class Monkey{
    .....
```

```

private String firstname;
private String lastname;

public String getName(){
    return firstname+ " "+lastname;
}
public void setName(String name){
    StringTokenizer t=new StringTokenizer(name);
    firstname=t.nextToken();
    lastname=t.nextToken();
}
.....
}

```

在 Monkey.hbm.xml 文件中,无须映射 Monkey 类的 firstname 和 lastname 属性,而是映射 name 属性,它和 MONKEYS 表的 NAME 字段对应。

```
<property name="name" column="NAME" />
```

尽管在 Monkey 类中并没有定义 name 属性,由于 Hibernate 并不会直接访问 name 属性,而是调用 getName()和 setName()方法,因此,实际上建立了 Monkey 类的 firstname 和 lastname 属性与 MONKEYS 表的 NAME 字段的映射,参见图 3-2。

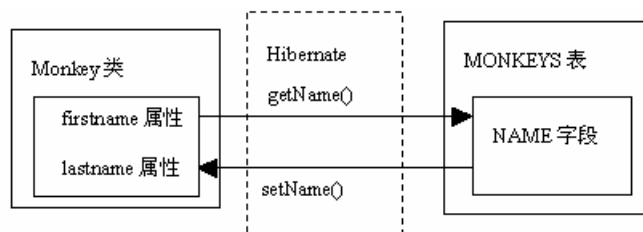


图 3-2 Monkey 类的 firstname 和 lastname 属性与 MONKEYS 表的 NAME 字段的映射

不管在 Monkey 类中是否存在 name 属性,只要在 Monkey.hbm.xml 文件中映射了 name 属性,在 HQL 语句中就能访问它:

```

Query query=session.createQuery
    ("from Monkey as m where m.name='悟空 孙' ");

```

尽管在 Monkey 类中定义了 firstname 属性,但是没有在 Monkey.hbm.xml 文件中映射 firstname 属性,因此以下 HQL 语句是不正确的:

```

Query query =session.createQuery
    ("from Monkey as m where m.firstname='悟空' ");

```

当 Hibernate 执行以上 HQL 语句时,会抛出以下错误信息:

```

org.hibernate.QueryException: could not resolve property: firstname of: mypack.Monkey
[from mypack.Monkey as m where m.firstname='悟空' ]

```

## 2. 在Monkey类的setGender()方法中加入数据验证逻辑

在持久化类的访问方法中，还可以加入数据验证逻辑。例如，以下代码在 setGender()方法中先判断参数 gender 是否为'F'或者'M'，如果不是，就抛出异常。

```
public void setGender(char gender){
    if(gender!='F' && gender!='M'){
        throw new IllegalArgumentException("Invalid Gender");
    }
    this.gender = gender ;
}
```

setGender()方法有两个调用者：Java 应用程序和 Hibernate。当 Java 应用程序调用 setGender(char gender)方法时，参数值很有可能来自于从用户界面输入的数据，因此有必要执行数据验证。而当 Hibernate 调用 setGender(char gender)方法时，参数值来自于从数据库读出的数据，通常都是合法数据，因此没必要执行数据验证。解决这一矛盾的办法是把 Monkey.hbm.xml 文件中映射 gender 属性的<property>元素的 access 属性设为“field”：

```
<property name="gender" access="field" column="GEDNER" />
```

这样，Hibernate 就会直接访问 Monkey 实例的 gender 属性，而不是调用 setGender()和 getGender()方法。

### Tips

在实际应用中，数据验证逻辑通常由表述层或者业务逻辑层来实现。在持久化类中加入数据验证逻辑，主要是便于在软件开发和测试阶段能捕获表述层或者业务逻辑层应该处理而未处理的异常，提醒开发人员在表述层或者业务逻辑层加入相关的数据验证逻辑。

### 3.1.4 设置派生属性

并不是持久化类的所有属性都直接和表的字段匹配，持久化类的有些属性的值必须在运行时通过计算才能得出来，这种属性称为派生属性。本章 3.1.3 节已经给出了一种解决方案，以 Monkey 类的 firstname 属性为例，该方案包括两个步骤。

(1) 在映射文件中不映射 firstname 属性，而是映射 name 属性。

(2) 在 Monkey 类的 setName()方法中加入程序逻辑，自动为 firstname 属性赋值。

本节给出另一种解决方案：利用<property>元素的 formula 属性。formula 属性用来设置一个 SQL 表达式，Hibernate 将根据它来计算出派生属性的值。假定 Monkey 类有一个 avgAge 属性，表示所有猴子的平均年龄，而在 MONKEYS 表中没有对应的 AVG\_AGE 字段。下面以映射 avgAge 属性为例，介绍<property>元素的 formula

属性的用法。在 Monkey.xml 文件中映射 avgAge 属性的代码如下：

```
<property name="avgAge"
  formula="(select avg(m.AGE) from MONKEYS m)" />
```

当 Hibernate 从数据库中查询 Monkey 对象时，在 select 语句中会包含以上用于计算 avgAge 派生属性的子查询语句：

```
select ID,NAME, GENDER, AGE,`MONKEY DESCRIPTION`,
(select avg(m.AGE) from MONKEYS m) from MONKEYS;
```

<property>元素的 formula 属性指定一个 SQL 表达式，该表达式可以引用表的字段，包含子查询语句或者调用 SQL 函数。例如：

```
<property name="email" formula="lower(EMAIL)" />
```

### 3.1.5 控制insert和update语句

Hibernate 在初始化阶段，就会根据映射文件的映射信息，为所有的持久化类预定义以下 SQL 语句：

- insert 语句，例如 Monkey 类的 insert 语句如下：

```
insert into MONKEY(ID,NAME,GENDER,AGE,`MONKEY DESCRIPTION`)
values(?,?,?,?,?)
```

- update 语句，例如 Monkey 类的 update 语句如下：

```
update MONKEYS set NAME=?, GENDER=?,AGE=?,`MONKEY DESCRIPTION`=? where
ID=?
```

- delete 语句，例如 Monkey 类的 delete 语句如下：

```
delete from MONKEYS where ID=?
```

- 根据 OID 来从数据库加载持久化类实例的 select 语句，例如 Monkey 类的 select 语句如下：

```
select ID,NAME,GENDER,AGE,`MONKEY DESCRIPTION` from MONKEYS where ID=?
```

以上 SQL 语句中的问号代表 JDBC PreparedStatement 中的参数。这些 SQL 语句都存放在 SessionFactory 的内置缓存中，当执行 Session 的 save()、update()、delete()、load()和 get()方法时，将从缓存中找到相应的预定义 SQL 语句，再把具体的参数值绑定到该 SQL 语句中。

在默认情况下，预定义的 SQL 语句中包含了表的所有字段。此外，Hibernate 还允许在映射文件中控制 insert 和 update 语句的内容，例如：

```
<property name="age" update="false" column="AGE" />
```

以上代码用于映射 Monkey 类的 age 属性，它的<property>元素的 update 属性

设为 false，这表明在 update 语句中不会包含 AGE 字段。表 3-2 列出了所有用于控制 insert 和 update 语句的映射属性，本章 3.4 节举例演示了这几个映射属性对 Hibernate 运行时行为的影响。

表 3-2 用于控制 insert 和 update 语句的映射属性

映 射 属 性	作 用
<property>元素的 insert 属性	如果为 false，在 insert 语句中不包含该字段，表明该字段永远不能被插入。默认值为 true
<property>元素的 update 属性	如果为 false，update 语句中不包含该字段，表明该字段永远不能被更新。默认值为 true
<class>元素的 mutable 属性	如果为 false，等价于所有的<property>元素的 update 属性为 false，表示整个实例不能被更新。默认值为 true
<class>元素的 dynamic-insert 属性	如果为 true，表示当保存一个对象时，会动态生成 insert 语句，insert 语句中仅包含所有取值不为 null 的字段。默认值为 false
<class>元素的 dynamic-update 属性	如果为 true，表示当更新一个对象时，会动态生成 update 语句，update 语句中仅包含所有取值需要更新的字段。默认值为 false

Hibernate 生成动态 SQL 语句的系统开销（如占用 CPU 的时间和占用的内存）很小，因此不会影响应用的运行性能。如果表中包含许多字段，建议把 dynamic-insert 属性和 dynamic-update 属性都设为 true。这样，在 insert 和 update 语句中就只包含需要插入或更新的字段，这可以节省数据库执行 SQL 语句的时间，从而提高应用的运行性能。

## 3.2 处理SQL引用标识符

在 SQL 语法中，标识符是指用于为数据库表、视图、字段或索引等命名的字符串，常规标识符不包含空格，也不包含特殊字符，因此无须使用引用符号，例如以下 SQL 语句中，MONKEYS、ID 和 NAME 都是标识符：

```
create table MONKEYS (
    ID bigint not null,
    NAME varchar(15) not null,
    .....
);
```

如果数据库表名或字段名中包含空格，或者包含特殊字符，那么可以使用引用标识符。在 MySQL 中，引用标识符的形式为`IDENTIFIER NAME`。例如，以下 SQL 语句创建的 MONKEYS 表中有一个引用标识符字段`MONKEY DESCRIPTION`：

```
create table MONKEYS (
```

```
ID bigint not null,
.....
`MONKEY DESCRIPTION` text
);
```

在映射 Monkey 类的 description 属性时, 也应该使用引用标识符, 例如:

```
<property name="description" column="`MONKEY DESCRIPTION`"/>
```

当 Hibernate 生成 SQL 语句时, 将始终采用引用标识符的形式, 来访问 `MONKEY DESCRIPTION` 字段。例如, 以下是 Hibernate 生成的 insert 语句的形式:

```
insert into MONKEYS (ID,NAME, GENDER, AGE,`MONKEY DESCRIPTION`)
values(?,?,?,?);
```

对于不同数据库系统, 引用标识符有不同的形式。在 MS SQL Server 中, 引用标识符的形式为 [IDENTIFIER NAME]; 在 MySQL 中, 引用标识符的形式为 `IDENTIFIER NAME`。但是在设置 Hibernate 的映射文件时, 可以一律采用 `IDENTIFIER NAME` 的形式。Hibernate 会自动根据 Hibernate 配置文件中设置的 SQL 方言 (即 hibernate.dialect 属性), 来生成正确的 SQL 语句。

### 3.3 使用XML格式的配置文件

Hibernate 的配置文件有两种形式: 一种是 XML 格式的文件; 还有一种是 Java 属性文件, 采用 “键=值” 的形式。本书第 2 章使用的是 Java 属性文件, 本章介绍 XML 格式的配置文件。XML 格式的配置文件的默认名字为 “hibernate.cfg.xml”, 这个文件也应该放在 classpath 的根路径下。

XML 格式的配置文件不仅能设置所有的 Hibernate 配置选项, 还能够通过 <mapping> 元素声明需要加载的映射文件, 参见例程 3-2。

例程 3-2 hibernate.cfg.xml 文件

```
<hibernate-configuration>
  <session-factory >
    <property name="dialect">org.hibernate.dialect.MySQLDialect
    </property>

    <property name="connection.driver_class">
      com.mysql.jdbc.Driver
    </property>

    <property name="connection.url">
      jdbc:mysql://localhost:3306/sampled
    </property>
```

```
<property name="connection.username">root</property>
<property name="connection.password">1234</property>
<property name="show_sql">true</property>

<mapping resource="mypack/Monkey.hbm.xml" />

</session-factory>
</hibernate-configuration>
```

以上<session-factory>元素的<mapping>子元素指定需要加载的映射文件。一个<session-factory>元素中可以包含多个<mapping>子元素。

如果 Hibernate 的配置文件为 Java 属性文件，那么 Java 程序必须以编程方式声明需要加载的映射文件：

```
SessionFactory sessionFactory = new Configuration()
    .addClass(mypack.Monkey.class)
    .buildSessionFactory();
```

如果 Hibernate 的配置文件为 XML 格式，那么只需在配置文件中声明映射文件，在程序中不必调用 Configuration 类的 addClass()方法来加载映射文件。当映射文件名发生变化，只需修改 XML 格式的配置文件，不需要修改程序代码，因此 XML 格式的配置文件会提高应用程序的可维护性。

当通过 Configuration 的默认构造方法 Configuration()来创建 Configuration 实例时，Hibernate 会到 classpath 中查找默认的 hibernate.properties 文件，如果找到，就把它的配置信息加载到内存中。在默认情况下，Hibernate 不会加载 hibernate.cfg.xml 文件，必须通过 Configuration 的 configure()方法来显式加载 hibernate.cfg.xml 文件：

```
SessionFactory sessionFactory = new Configuration()
    .configure() //加载 hibernate.cfg.xml 文件
    .buildSessionFactory();
```

Configuration 的 configure()方法会到 classpath 中查找 hibernate.cfg.xml 文件，如果找到，就把它的配置信息加载到内存中，如果没有找到该文件，会抛出异常：

```
org.hibernate.HibernateException: /hibernate.cfg.xml not found
```

### 3.4 运行本章的范例程序

本章范例程序位于配套光盘的 sourcecode\chapter3 目录下，它用于演示本章介绍的各种映射技巧。例程 3-3 和例程 3-4 分别是 Monkey.java 和 Monkey.hbm.xml 的源代码，例程中的粗体字部分为需要特别注意的地方，在这些地方都给出了详细的中文注解。



例程 3-3 Monkey.java

```
package mypack;

import java.util.*;

public class Monkey{

    private Long id;
    private String firstname;
    private String lastname;
    private char gender;
    private int age;
    private int avgAge;
    private String description;

    public Monkey() {}

    public Monkey(String firstname,String lastname,
        char gender,int age,String description){

        this.firstname=firstname;
        this.lastname=lastname;
        this.gender=gender;
        this.age=age;
        this.description=description;
    }

    public Long getId() {
        return this.id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public String getFirstname(){
        return firstname;
    }

    public void setFirstname(String firstname){
        this.firstname=firstname;
    }

    public String getLastname(){
        return lastname;
    }
}
```

```

public void setLastname(String lastname){
    this.lastname = lastname;
}

public String getName(){
    //加入处理逻辑
    return firstname+ " "+lastname;
}

public void setName(String name){
    //加入处理逻辑
    StringTokenizer t=new StringTokenizer(name);
    firstname=t.nextToken();
    lastname=t.nextToken();
}

public int getAge(){
    return this.age;
}

public void setAge( int age ){
    this.age = age;
}

public int getAvgAge(){
    return this.avgAge;
}

/** setAvgAge()方法仅被 Hibernate 访问 */
private void setAvgAge( int avgAge){
    this.avgAge = avgAge;
}

public char getGender(){
    return this.gender;
}

public void setGender(char gender){
    if(gender!='F' && gender!='M'){ /** 加入数据验证逻辑*/
        throw new IllegalArgumentException("Invalid Gender");
    }
    this.gender =gender ;
}

public String getDescription(){

```

```

        return this.description;
    }

    public void setDescription(String description){
        this.description=description;
    }
}

```

例程 3-4 Monkey.hbm.xml

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

    <!-- 设置了 dynamic-insert 和 dynamic-update 属性,
    表示会动态生成 MONKEYS 表的 insert 和 update 语句 -->
    <class name="mypack.Monkey" table="MONKEYS"
    dynamic-insert="true" dynamic-update="true" >

        <id name="id">
            <generator class="increment"/>
        </id>

        <property name="name" column="NAME" />

        <!-- 把 access 属性设为 field, 因此 Hibernate 会直接访问 gender 属性,
        而不会调用 getGender() 和 setGender() 方法,
        这可以避免 Hibernate 执行 setGender() 方法中的数据验证逻辑 -->
        <property name="gender" column="GENDER" access="field" />

        <property name="age" column="AGE" />

        <!-- avgAge 为派生属性 -->
        <property name="avgAge"
            formula="(select avg(m.AGE) from MONKEYS m)" />

        <!-- description 属性对应的字段名中有引用标识符 -->
        <property name="description" type="text"
            column="`MONKEY DESCRIPTION`" />

    </class>
</hibernate-mapping>

```

在 DOS 命令行下进入 chapter3 根目录, 然后输入命令: ant run, 就会运行

BusinessService 类。例程 3-5 是 BusinessService 类的源程序。

例程 3-5 BusinessService.java

```
package mypack;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class BusinessService{
    public static SessionFactory sessionFactory;
    static{
        try{
            Configuration config = new Configuration()
                .configure();//加载hibernate.cfg.xml文件中配置的信息
            sessionFactory = config.buildSessionFactory();
        }catch(RuntimeException e){e.printStackTrace();throw e;}
    }

    public Monkey loadMonkey(long monkey_id){.....}
    public void saveMonkey(Monkey monkey){.....}
    public void loadAndUpdateMonkey(long monkeyId){.....}
    public void updateMonkey(Monkey monkey){.....}

    public void printMonkey(Monkey monkey){.....}

    public void test(){
        Monkey monkey=new Monkey("悟空","孙",'M',500,"神通广大!");
        saveMonkey(monkey);

        monkey=loadMonkey(1);
        printMonkey(monkey);

        monkey.setDescription("齐天大圣!");
        updateMonkey(monkey);

        loadAndUpdateMonkey(1);
    }

    public static void main(String args[]){
        new BusinessService().test();
        sessionFactory.close();
    }
}
```

BusinessService 的 main()方法调用 test()方法，test()方法执行以下步骤。

(1) 保存一个所有属性都不为 null 的 Monkey 对象：

```
Monkey monkey=new Monkey("悟空","孙",'M',500,"神通广大!");
saveMonkey(monkey);
```

当 Session 的 save()方法保存以上 Monkey 对象时, Session 执行的 insert 语句为:

```
insert into MONKEYS (ID,NAME, GENDER, AGE,`MONKEY DESCRIPTION`)
values (1,'悟空 孙','M',500,'神通`大!');
```

(2) 加载并打印 OID 为 1 的 Monkey 对象:

```
monkey=loadMonkey(1);
printMonkey(monkey);
```

当加载 Monkey 对象时, 为了计算 avgAge 派生属性的值, 在 select 语句中包含子查询语句:

```
select ID,NAME, GENDER, AGE,`MONKEY DESCRIPTION`,
(select avg(m.AGE) from MONKEYS m) from MONKEYS
where ID=1;
```

printMonkey()方法的打印结果如下:

```
[java] name:悟空 孙
[java] gender:M
[java] description: 神通`大!
[java] age:500
[java] avgAge:500
```

(3) 修改 Monkey 对象的 description 属性, 然后更新 Monkey 对象:

```
monkey.setDescription("齐天大圣!");
updateMonkey(monkey);
```

updateMonkey()方法的代码如下:

```
tx = session.beginTransaction();
session.update(monkey);
tx.commit();
```

在运行以上 Session 的 update()方法时, Hibernate 执行以下 update 语句:

```
update MONKEYS set NAME='悟空 孙', GENDER='M', AGE=500,
`MONKEY DESCRIPTION`='齐天大圣!' where ID=1;
```

尽管 Monkey.hbm.xml 文件中<class>元素的 dynamic-update 属性为 true, 但是在动态生成的 update 语句中仍然包含所有的字段, 这是为什么呢? 这是因为对于以上 updateMonkey()方法, 在 Session 的缓存中没有原先 Monkey 对象的快照 (Snapshot, 即 Monkey 对象的复制), 因此无法判断当前 Monkey 对象的哪些属性被修改, 哪些属性没有被修改, 所以只能在 update 语句中包含所有的字段。如果读者对此难以理解, 可以在阅读完第 6 章 (通过 Hibernate 操纵对象) 后再来回顾

这段内容。

(4) 加载并修改 OID 为 1 的 Monkey 对象：

```
loadAndUpdateMonkey(1);
```

loadAndUpdateMonkey()方法的代码如下：

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(monkeyId));
monkey.setDescription("勇敢无畏!");
tx.commit();
```

当 Session 加载了 Monkey 对象后，会在缓存中生成该 Monkey 对象的快照，通过比较当前 Monkey 对象与它的快照，Hibernate 能够判断当前 Monkey 对象的哪些属性被修改，哪些属性没有被修改，因此在 update 语句中只会包含需要更新的字段，Hibernate 执行的 update 语句为：

```
update MONKEYS set `MONKEY DESCRIPTION`='勇敢无畏!' where ID=1;
```

## 3.5 小结

本章主要介绍了单个持久化类与单个数据库表之间进行映射的方法，尤其是当持久化类的属性不和数据库表的字段一一对应时的映射技巧。也许您会问，为什么不将持久化类的属性和数据库表的字段都直接对应，从而简化两者之间的映射过程呢？对这个问题有以下三点解释。

(1) 对象模型和关系数据模型都是各自独立设计出来的，对象模型和关系数据模型有不同的设计原则，不能强迫某一方放弃本身的设计原则，从而和另一方建立直接的对应关系。举例来说，假如经常有这样的业务需求：查询猴子的平均年龄。为了方便业务逻辑层的编程，不妨在 Monkey 类中定义一个 avgAge 属性，每次从数据库中查询 Monkey 对象时，Hibernate 都会自动为 avgAge 属性赋值。如果在 Monkey 类中没有这个属性，业务逻辑层查询出 Monkey 对象后，还必须再单独到数据库中查询所有猴子的平均年龄。

另一方面，在 MONKEYS 表中没有必要提供 AVG\_AGE 字段，关系数据库中存放的是永久数据，应该尽量避免数据的冗余。为什么称 AVG\_AGE 是冗余字段呢？因为它的值可以根据 MONKEYS 表的 AGE 字段计算出来。冗余字段有两个缺点：一是浪费数据库存储空间；二是增加维护数据库的难度，如果修改了 MONKEYS 表的 AGE 字段，就必须同时修改 MONKEYS 表中的 AVG\_AGE 字段。

(2) 有的软件项目可能不是从头开发，而是建立在遗留的关系数据模型或对象模型的基础上，两种模型已经存在不对应之处，例如 Monkey 类中有 firstname

和 `lastname` 属性，而在 `MONKEYS` 表中只有 `NAME` 字段。由于升级或维护的困难，无法修改这两种模型的不对应之处。

(3) 假如软件项目是从头开发，在不违反对象模型和关系数据模型各自的设计原则的前提下，不妨尽量让它们保持直接的对应关系，这可以简化映射工作。例如 `Monkey` 类中有 `firstname` 和 `lastname` 属性，那么在 `MONKEYS` 表中提供 `FIRSTNAME` 和 `LASTNAME` 字段。

1

2

3

4

5

6

7

8

## 第 4 章 映射对象标识符

以下程序代码从数据库中加载 OID 为 1 的 Monkey 对象：

```
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
```

执行完上述代码后，Session 缓存中 OID 为 1 的 Monkey 对象与 MONKEYS 表中 ID 主键为 1 的记录对应。

以下程序代码两次从数据库中加载 OID 为 1 的 Monkey 对象，然后又分别修改两次加载的 Monkey 对象的 age 属性：

```
Transaction tx=session.beginTransaction();
Monkey monkey1=(Monkey)session.get(Monkey.class,new
    Long(1));
Monkey monkey2=(Monkey)session.get(Monkey.class,new
    Long(1));
monkey1.setAge(2);
monkey2.setAge(3);
tx.commit(); //提交事务
```

假设 monkey1 变量和 monkey2 变量分别引用不同的 Monkey 对象，那就意味着内存中有两个 OID 为 1 的 Monkey 对象，并且它们都与 MONKEYS 表中 ID 主键为 1 的一条记录对应。当提交事务时，Hibernate 遇到了难题，无法决定按照哪个 Monkey 对象的属性变化来同步更新数据库。幸运的是，Hibernate 会保证在一个 Session 对象的缓存中，每个 Monkey 对象都具有唯一的 OID。以上程序代码中的 monkey1 变量和 monkey2 变量实际上引用的是同一个 Monkey 对象，所以不会出现假释中遇到的难题。

Java 语言按内存地址来识别和区分同一个类的不同对象，而关系数据库按主键值来识别和区分同一个表的不同记录。Hibernate 使用对象标识符（OID）来建立 Session 缓存中的对象和数据库表中记录的一一对应关系，对象的 OID 和数据库表的主键对应。

对于在程序中新建的对象，为了保证 OID 的唯一性和不可变性，应该让 Hibernate，而不是应用程序来为 OID 赋值。Hibernate 通过标识符生成器来为 OID 赋值。例如对于以下程序代码：

```
Monkey monkey=new Monkey();
monkey.setName("智多星");
monkey.setAge(1);
monkey.setGender('M');

session.save(monkey);
```



```
System.out.println(monkey.getId()); //打印一个具体的OID值，而不是null。
```

以上程序代码并没有为 **Monkey** 对象设置 **OID**，但是当调用 `session.save(monkey)` 方法时，**Hibernate** 会根据对象-关系映射文件的配置来为 **Monkey** 对象设置 **OID**。假定在 **Monkey.hbm.xml** 文件中相关的映射代码如下：

```
<id name="id" column="ID" type="long">
  <generator class="increment"/>
</id>
```

那么 **Hibernate** 就会通过 **increment** 标识符生成器来为 **Monkey** 对象设置 **OID**。假定把 **OID** 设置为 1，那么 **Hibernate** 会向 **MONKEYS** 表中插入一条 **ID** 为 1 的记录。

本章主要介绍 **Hibernate** 提供的几种内置标识符生成器的用法。本章最后的 4.5 节介绍了映射自然主键的方法，由于这一节的内容涉及游离对象和临时对象等概念，建议在阅读了第 6 章（通过 **Hibernate** 操纵对象）内容后再来看这一节。

## 4.1 关系数据库按主键区分不同的记录

在关系数据库表中，用主键来识别记录并保证每条记录的唯一性。作为主键的字段必须满足以下条件：

- 不允许为 **null**。
- 每条记录具有唯一的主键值，不允许主键值重复。
- 每条记录的主键值永远不会改变。

在 **MONKEYS** 表中，如果把 **NAME** 字段作为主键，前提条件是：

- 每条记录的猴子姓名不允许为 **null**。
- 不允许猴子重名。
- 不允许修改猴子姓名。

**NAME** 字段是具有业务含义的字段，把这种字段作为主键，称为自然主键。尽管也是可行的，但是不能满足不断变化的业务需求，一旦出现了允许猴子重名的业务需求，就必须修改数据模型，重新定义表的主键，这给数据库的维护增加了难度。

因此，更合理的方式是使用代理主键，即不具备业务含义的字段，该字段一般取名为“**ID**”。代理主键通常为整数类型，因为整数类型比字符串类型要节省更多的数据库空间。那么代理主键的值从何而来呢？许多数据库系统都提供了自动生成代理主键值的机制。下面举例说明。

### 4.1.1 把主键定义为自动增长标识符类型

在 MySQL 中，如果把表的主键设为 `auto_increment` 类型，数据库就会自动为主键赋值。例如：

```
create table MONKEYS(ID int auto_increment primary key not null, NAME varchar(15));
insert into MONKEYS (NAME) values("Tom");
insert into MONKEYS (NAME) values("Mike");
select ID,NAME from MONKEYS;
```

以上 SQL 语句先创建了 MONKEYS 表，然后插入两条记录，在插入时仅仅设定了 NAME 字段的值。最后查询 MONKEYS 表中的记录，查询结果为：

ID	NAME
1	Tom
2	Mike

由此可见，一旦把 ID 字段设为 `auto_increment` 类型，MySQL 数据库会自动按递增的方式为主键赋值。

在 MS SQL Server 中，如果把表的主键设为 `identity` 类型，数据库就会自动为主键赋值。例如：

```
create table MONKEYS(ID int identity(1,1) primary key not null, NAME varchar(15));
insert into MONKEYS (NAME) values("Tom");
insert into MONKEYS (NAME) values("Mike");
select ID, NAME from MONKEYS;
```

在 MS SQL Server 中执行以上 SQL 语句，最后查询结果为：

ID	NAME
1	Tom
2	Mike

由此可见，一旦把 ID 字段设为 `identity` 类型，MS SQL Server 数据库会自动按递增的方式为主键赋值。`identity` 包含两个参数，第一个参数表示起始值，第二个参数表示增量。

### 4.1.2 从序列 (Sequence) 中获取自动增长的标识符

在 Oracle 数据库系统中，可以为每张表的主键创建一个单独的序列，然后从这个序列中获得自动增加的标识符，把它赋值给主键。例如以下语句创建了一个名为 MONKEYS\_ID\_SEQ 的序列，这个序列的起始值为 1，增量为 2。

```
create sequence MONKEYS_ID_SEQ increment by 2 start with 1
```

一旦定义了 MONKEYS\_ID\_SEQ 序列，就可以访问序列的 `curval` 和 `nextval` 属性。

- curval: 返回序列的当前值。
- nextval: 先增加序列的值, 然后返回增加后的序列值。

以下 SQL 语句先创建了 MONKEYS 表, 然后插入两条记录, 在插入时设定了 ID 和 NAME 字段的值, 其中 ID 字段的值来自于 MONKEYS\_ID\_SEQ 序列。最后查询 MONKEYS 表中的记录。

```
create table MONKEYS (ID int primary key not null, NAME varchar(15));
insert into MONKEYS values(MONKEYS_ID_SEQ.curval, 'Tom');
insert into MONKEYS values(MONKEYS_ID_SEQ.nextval, 'Mike');
select ID,NAME from monkeys;
```

如果在 Oracle 中执行以上 SQL 语句, 查询结果为:

ID	NAME
1	Tom
3	Mike

## 4.2 Java语言按内存地址区分不同的对象

在 Java 语言中, 判断两个对象引用变量是否相等, 有以下两种比较方式。

(1) 比较两个变量所引用的对象的内存地址是否相同, “==” 运算符就是比较的内存地址。此外, 在 Object 类中定义的 equals(Object o)方法, 也是按内存地址来比较的。如果用户自定义的类没有覆盖 Object 类的 equals(Object o)方法, 也按内存地址比较。例如, 以下代码用 new 语句共创建了两个 Monkey 对象, 并定义了 3 个 Monkey 类型的引用变量 m1、m2 和 m3:

```
Monkey m1=new Monkey("Tom"); //line1
Monkey m2=new Monkey("Tom"); //line2
Monkey m3=m1; //line3
m3.setName("Mike"); //line4
```

图 4-1 和图 4-2 显示了程序执行到第 3 行及第 4 行的对象图。

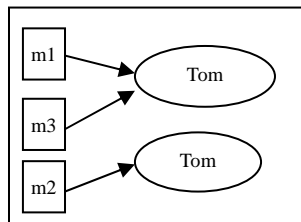


图 4-1 程序执行到第 3 行的对象图

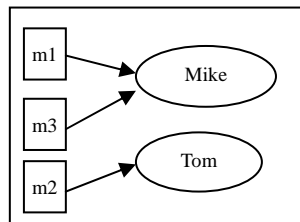


图 4-2 程序执行到第 4 行的对象图

从图 4-1 和图 4-2 看出, m1 和 m3 变量引用同一个 Monkey 对象, 而 m2 变量引用另一个 Monkey 对象。因此, 表达式 “m1 == m3” 及 m1.equals(m3)的值都是

true, 而表达式 “m1 == m2” 及 m1.equals(m2) 的值都是 false。

(2) 比较两个变量所引用的对象的值是否相同, Java API 中的一些类覆盖了 Object 类的 equals(Object o) 方法, 实现按对象值比较。这些类包括:

- String 类和 Date 类。
- Java 包装类, 包括: Byte、Integer、Short、Character、Long、Float、Double 和 Boolean。

例如:

```
String s1=new String("hello");
String s2=new String("hello");
```

尽管 s1 和 s2 引用不同的 String 对象, 但是它们的字符串值都是 “hello”, 因此表达式 “s1 == s2” 的值是 false, 而表达式 s1.equals(s2) 的值是 true。

用户自定义的类也可以覆盖 Object 类的 equals(Object o) 方法, 从而实现按对象值比较。例如, 在 Monkey 类中添加如下 equals(Object o) 方法, 使它按猴子的姓名来比较两个 Monkey 对象是否相等:

```
public boolean equals(Object o){
    if(this==o)return true;
    if (!o instanceof Monkey)
        return false;
    final Monkey other=(Monkey)o;
    if(this.getName().equals(other.getName()))
        return true;
    else
        return false;
}
```

以下代码用 new 语句共创建了两个 Monkey 对象, 并定义了两个 Monkey 类型的引用变量 m1 和 m2:

```
Monkey m1=new Monkey("Tom");
Monkey m2=new Monkey("Tom");
```

尽管 m1 和 m2 引用不同的 Monkey 对象, 但是它们的 name 值都是 “Tom”, 因此表达式 “m1 == m2” 的值是 false, 而表达式 m1.equals(m2) 的值是 true。

### 4.3 Hibernate 用对象标识符 (OID) 来区分对象

从以上两节可以看出, Java 语言按内存地址来识别或区分同一个类的不同对象, 而关系数据库按主键值来识别或区分同一个表的不同记录。Hibernate 使用 OID 来统一两者之间的矛盾, OID 是关系数据库中的主键 (通常为代理主键) 在 Java

对象模型中的等价物。在运行时，Hibernate 根据 OID 来维持 Java 对象和数据库表中记录的对应关系。例如：

```
Transaction tx = session.beginTransaction(); //line1
Monkey m1=(Monkey)session.get(Monkey.class, new Long(1)); //line2
Monkey m2=(Monkey)session.get(Monkey.class, new Long(1)); //line3
Monkey m3=(Monkey)session.get(Monkey.class,new Long(3)); //line4
System.out.println(m1==m2); //line5
System.out.println(m1==m3); //line6

tx.commit();
```

在以上程序中，3 次调用了 Session 的 get()方法，分别加载 OID 为 1 或 3 的 Monkey 对象。以下是 Hibernate 3 次加载 Monkey 对象的流程。

(1) 第一次加载 OID 为 1 的 Monkey 对象时，先从数据库的 MONKEYS 表中查询 ID 为 1 的记录，再创建相应的 Monkey 实例，把它保存在 Session 缓存中，最后把这个对象的引用赋值给变量 m1。

(2) 第二次加载 OID 为 1 的 Monkey 对象时，直接把 Session 缓存中 OID 为 1 的 Monkey 对象的引用赋值给 m2，因此 m1 和 m2 引用同一个 Monkey 对象。

(3) 当加载 OID 为 3 的 Monkey 对象时，由于在 Session 缓存中还不存在这样的对象，所以必须再次到数据库中查询 ID 为 3 的记录，再创建相应的 Monkey 实例，把它保存在 Session 缓存中，最后把这个对象的引用赋值给变量 m3。

因此，表达式“m1==m2”的结果为 true，表达式“m1==m3”的结果为 false。图 4-3 显示了 Session 缓存中的 Monkey 对象和 MONKEYS 表中记录的对应关系。

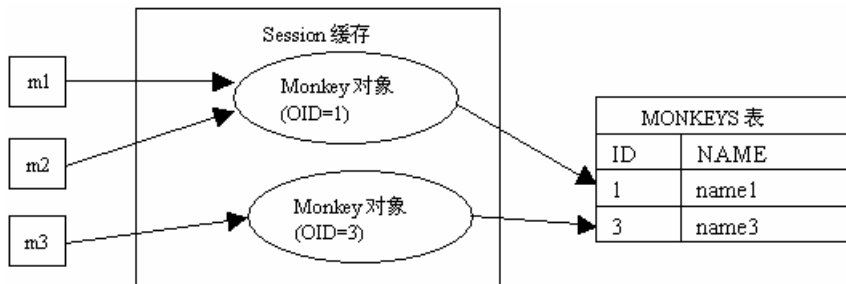


图 4-3 Session 缓存中的 Monkey 对象和 MONKEYS 表中记录的对应关系

与表的代理主键对应，OID 也是整数类型，Hibernate 允许在持久化类中把 OID 定义为以下整数类型。

- short (或包装类 Short): 2 个字节，取值范围是：-2<sup>15</sup> ~ 2<sup>15</sup>-1
- int (或包装类 Integer): 4 个字节，取值范围是：-2<sup>31</sup> ~ 2<sup>31</sup>-1
- long (或包装类 Long): 8 个字节，取值范围是：-2<sup>63</sup> ~ 2<sup>63</sup>-1

为了保证持久化对象的 OID 的唯一性和不可变性，通常由 Hibernate 或底层数据库来给 OID 赋值。因此，可以把持久化类的 OID 的 setId()方法设为 private 类型，以禁止 Java 应用程序随便修改 OID。而把 getId()方法设为 public 类型，这使得 Java 应用程序可以读取持久化对象的 OID：

```
private Long id;
private void setId(Long id){
    this.id=id;
}
public Long getId(){
    return id;
}
```

在对象-关系映射文件中，<id>元素用来设置对象标识符，例如：

```
<id name="id" type="long" column="ID">
    <generator class="increment"/>
</id>
```

<generator>子元素用来设定标识符生成器。Hibernate 提供了标识符生成器接口，即 org.hibernate.id.IdentifierGenerator 接口，并且提供了多种内置的实现。例如 org.hibernate.id.IdentityGenerator 和 org.hibernate.id.IncrementGenerator 为两种内置实现类，它们的缩写名分别为“identity”和“increment”。在设置<generator>子元素的 class 属性时，既可以提供完整的标识符生成器的类名，也可以给定缩写名，因此以下两种配置方式是等价的：

```
<id name="id" type="long" column="ID">
    <generator class="increment"/>
</id>
<!-- 或者 -->
<id name="id" type="long" column="ID">
    <generator class="org.hibernate.id.IncrementGenerator"/>
</id>
```

表 4-1 列出了 Hibernate 提供的几种内置标识符生成器，下面一节将详细介绍一些常用标识符生成器的用法。

表 4-1 Hibernate 提供的内置标识符生成器

标识符生成器	描 述
increment	适用于代理主键。由 Hibernate 自动以递增的方式生成标识符，每次增量为 1
identity	适用于代理主键。由底层数据库生成标识符。前提条件是底层数据库支持自动增长字段类型，例如 DB2, MySQL, MS SQL Server, Sybase 和 HypersonicSQL
sequence	适用于代理主键。Hibernate 根据底层数据库的序列来生成标识符。前提条件是底层数据库支持序列，例如 DB2, PostgreSQL, Oracle 和 SAP DB

(续表)

标识符生成器	描 述
hilo	适用于代理主键。Hibernate 根据 high/low 算法来生成标识符。Hibernate 把特定表的字段作为“high”值。在默认情况下选用 hibernate_unique_key 表的 next_hi 字段
native	适用于代理主键。根据底层数据库对自动生成标识符的支持能力, 来选择 identity、sequence 或 hilo
uuid.hex	适用于代理主键。Hibernate 采用 128 位的 UUID(Universal Unique Identification)算法来生成标识符。UUID 算法能够在网络环境中生成唯一的字符串标识符。这种标识符生成策略并不流行, 因为字符串类型的主键比整数类型的主键占用更多的数据库空间
assigned	适用于自然主键。由 Java 应用程序负责生成标识符, 为了能让 Java 应用程序设置 OID, 不能把 setId()方法声明为 private 类型。应该尽量避免使用自然主键
select	适用于遗留数据库中的代理主键或自然主键。由数据库中的触发器来生成标识符
foreign	用另一个关联的对象的标识符来作为当前对象的标识符, 主要适用于一对一关联的场合

## 4.4 Hibernate的内置标识符生成器的用法

本节结合具体例子来演示几种常用标识符生成器的用法。本范例提供了以下映射文件:

- IncrementTester.hbm.xml: 把 IncrementTester 持久化类映射到 INCREMENT\_TESTER 表。用于演示 increment 标识符生成器的用法。
- IdentityTester.hbm.xml: 把 IdentityTester 持久化类映射到 IDENTITY\_TESTER 表。用于演示 identity 标识符生成器的用法。
- SequenceTester.hbm.xml: 把 SequenceTester 持久化类映射到 SEQUENCE\_TESTER 表。用于演示 sequence 标识符生成器的用法。
- NativeTester.hbm.xml: 把 NativeTester 持久化类映射到 NATIVE\_TESTER 表。用于演示 native 标识符生成器的用法。
- HiloTester.hbm.xml: 把 HiloTester 持久化类映射到 HILO\_TESTER 表。用于演示 hilo 标识符生成器的用法。

IncrementTester、IdentityTester、SequenceTester、NativeTester 和 HiloTester 持久化类的代码很相似, 都有一个 id 属性和 name 属性, 以及相应的 get 和 set 方法。以下例程 4-1 为 IncrementTester 类的源程序。

例程 4-1 IncrementTester.java

```
package mypack;
public class IncrementTester {
    private Long id;
    private String name;
```

```

public IncrementTester() {}

public IncrementTester(String name) {
    this.name = name;
}

public Long getId() {
    return this.id;
}

private void setId(Long id) {
    this.id = id;
}

public String getName() {
    return this.name;
}

public void setName(String name) {
    this.name = name;
}
}

```

本范例还提供了—个 **BusinessService** 类，用来测试各种标识符生成器的用法。它的源程序参见例程 4-2。

#### 例程 4-2 BusinessService 类

```

package mypack;

import java.lang.reflect.Constructor;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.io.*;
import java.sql.Time;
import java.util.*;

public class BusinessService{
    public static SessionFactory sessionFactory;
    static{
        try{
            Configuration config = new Configuration().configure();
            sessionFactory = config.buildSessionFactory();
        }catch(RuntimeException e){e.printStackTrace();throw e;}
    }

    public void findAllObjects(String className){

```



```

Session session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    List objects=session.createQuery("from " +className).list();
    for (Iterator it = objects.iterator(); it.hasNext();) {
        Long id=new Long(0);
        if(className.equals("mypack.NativeTester"))
            id=((NativeTester) it.next()).getId();
        if(className.equals("mypack.IncrementTester"))
            id=((IncrementTester) it.next()).getId();
        if(className.equals("mypack.IdentityTester"))
            id=((IdentityTester) it.next()).getId();
        if(className.equals("mypack.HiloTester"))
            id=((HiloTester) it.next()).getId();

        System.out.println("ID of "+ className+":"+id);
    }

    tx.commit();
} catch (RuntimeException e) {
    if (tx != null) {
        tx.rollback();
    }
    throw e;
} finally {
    session.close();
}
}

public void saveObject(Object object){
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.save(object);
        tx.commit();

    } catch (RuntimeException e) {
        if (tx != null) {
            tx.rollback();
        }
        throw e;
    } finally {
        session.close();
    }
}

```

```

    }

    public void deleteAllObjects(String className){
        Session session = sessionFactory.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Query query=session.createQuery("delete from " +className);
            query.executeUpdate();
            tx.commit();

        }catch (RuntimeException e) {
            if (tx != null) {
                tx.rollback();
            }
            throw e;
        } finally {
            session.close();
        }
    }

    public void test(String className) throws Exception{
        deleteAllObjects(className);
        Object o1=Class.forName(className).newInstance();
        saveObject(o1);
        Object o2=Class.forName(className).newInstance();
        saveObject(o2);
        Object o3=Class.forName(className).newInstance();
        saveObject(o3);
        findAllObjects(className);
    }

    public static void main(String args[])throws Exception {
        String className;
        if(args.length==0)
            className="mypack.NativeTester";
        else
            className=args[0];
        new BusinessService().test(className);

        sessionFactory.close();
    }
}

```

在 `BusinessService` 类的 `test(String className)` 方法中，按照参数 `className` 给定的类名，先删除与这个类映射的表中的所有记录，然后创建这个类的 3 个对象，

把它们持久化到表中，最后把表中所有记录的 ID 打印出来。由源程序可以看出，在 `test(String className)` 方法中并没有为这些对象设置 OID。参数 `className` 的可选值包括：

- `mypack.IncrementTester`
- `mypack.IdentityTester`
- `mypack.NativeTester`
- `mypack.HiloTester`

本章范例位于配套光盘的 `sourcecode/chapter4` 目录下。可以利用 ANT 工具来运行 `BusinessService` 类，在范例程序的根目录 `chapter4` 下提供了 `build.xml` 程序，它定义了 `run_native`、`run_increment`、`run_identity` 和 `run_hilo` target。`run_increment` target 的定义如下，其中 `<arg>` 子元素用于设定 `BusinessService` 类的 `main()` 方法的参数：

```
<target name="run_increment" description="Run a Hibernate sample"
        depends="compile">
  <java classname="mypack.BusinessService" fork="true">
    <arg value="mypack.IncrementTester" />
    <classpath refid="project.class.path"/>
  </java>
</target>
```

在 DOS 命令行下，转到范例程序的根目录 `chapter4`，输入命令：`ant run_increment`，就会执行 `run_increment` target。

### 4.4.1 increment标识符生成器

`increment` 标识符生成器由 `Hibernate` 以递增的方式为代理主键赋值。例如，在 `IncrementTester.hbm.xml` 文件中声明使用 `increment` 标识符生成器：

```
<hibernate-mapping>
  <class name="mypack.IncrementTester" table="INCREMENT_TESTER">

    <id name="id" type="long" column="ID">
      <meta attribute="scope-set">private</meta>
      <generator class="increment"/>
    </id>

    <property name="name" type="string" column="NAME" />

  </class>
</hibernate-mapping>
```

数据库中 INCREMENT\_TESTER 表的 DDL 定义如下：

```
create table INCREMENT_TESTER (
    ID bigint not null,
    NAME varchar(15) not null,
    primary key (id)
);
```

在 DOS 下运行 `ant run_increment`，以下是在控制台输出的部分信息：

```
[java] insert into INCREMENT_TESTER (NAME, ID) values (?, ?)
[java] insert into INCREMENT_TESTER (NAME, ID) values (?, ?)
[java] insert into INCREMENT_TESTER (NAME, ID) values (?, ?)
[java] ID of mypack.IncrementTester:1
[java] ID of mypack.IncrementTester:2
[java] ID of mypack.IncrementTester:3
```

在以上 `insert` 语句中包含 `ID` 字段，由此可见，Hibernate 在持久化一个 `IncrementTester` 对象时，会以递增的方式自动生成标识符。事实上，Hibernate 会先读取 `INCREMENT_TESTER` 表中的最大主键值：

```
select max(ID) from INCREMENT_TESTER;
```

接下来向 `INCREMENT_TESTER` 表中插入记录时，就在 `max(ID)` 的基础上递增，增量为 1。下面考虑有两个 Hibernate 应用进程向同一个数据库的同一张表插入记录的情景。

(1) 假定第一个进程中的 Hibernate 先读取 `INCREMENT_TESTER` 表中的最大主键值为 6。

(2) 接着第二个进程中的 Hibernate 也读取 `INCREMENT_TESTER` 表中的最大主键值仍然为 6。

(3) 接下来两个进程中的 Hibernate 各自向 `INCREMENT_TESTER` 表中插入主键值为 7 的记录，这违反了数据库的完整性约束，导致有一个进程中的插入操作失败。

由此可见，`increment` 标识符生成器仅仅在只有单个 Hibernate 应用进程访问数据库的情况下才能有效工作。更确切地说，即使在同一个进程中创建了连接同一个数据库的多个 `SessionFactory` 实例，也会导致插入操作失败。在 JavaEE 软件架构中，Hibernate 通常作为 JNDI 资源运行在应用服务器上。如图 4-4 所示，如果 Hibernate 仅运行在单个应用服务器上，`increment` 标识符生成器能有效工作；如图 4-5 所示，如果 Hibernate 运行在多个应用服务器上（即在集群环境下），`increment` 标识符生成器工作会失效。

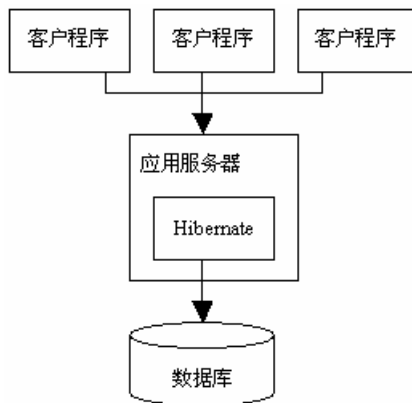


图 4-4 Hibernate 运行在单个应用服务器上

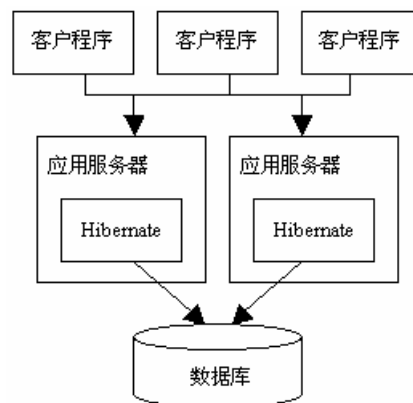


图 4-5 Hibernate 运行在多个应用服务器上

increment 标识符生成器具有以下适用范围。

- 由于 increment 生成标识符的机制不依赖于底层数据库系统，因此它适合于所有的数据库系统。
- 适用于只有单个 Hibernate 应用进程访问同一个数据库的场合，在集群环境下不推荐使用它。
- OID 必须为 long、int 或 short 类型，如果把 OID 定义为 byte 类型，在运行时抛出如下异常：

```
[java] org.hibernate.id.IdentifierGenerationException: this id generator
generates long, integer, short identity
```

#### 4.4.2 identity 标识符生成器

identity 标识符生成器由底层数据库来负责生成标识符，它要求底层数据库把主键定义为自动增长字段类型。例如，在 MySQL 中，应该把主键定义为 auto\_increment 类型；在 MS SQL Server 中，应该把主键定义为 identity 类型。在 IdentityTester.hbm.xml 文件中声明使用 identity 标识符生成器：

```
<hibernate-mapping>
  <class name="mypack.IdentityTester" table="IDENTITY_TESTER">

    <id name="id" type="long" column="ID">
      <meta attribute="scope-set">private</meta>
      <generator class="identity"/>
    </id>

    <property name="name" type="string" column="NAME" />

  </class>
</hibernate-mapping>
```

如果 Hibernate 连接的是 MySQL 数据库，那么数据库中 IDENTITY\_TESTER 表的 DDL 定义如下：

```
create table IDENTITY_TESTER (
    ID bigint not null auto_increment,
    NAME varchar(15) not null,
    primary key (id)
);
```

如果 Hibernate 连接的是 MS SQL Server，那么数据库中 IDENTITY\_TESTER 表的 DDL 定义如下：

```
create table NATIVE_TESTER (
    ID bigint not null identity,
    NAME varchar(15) not null,
    primary key (id)
);
```

在 DOS 下运行 `ant run_identity`，以下是在控制台输出的部分信息：

```
[java] insert into IDENTITY_TESTER (NAME) values (?)
[java] insert into IDENTITY_TESTER (NAME) values (?)
[java] insert into IDENTITY_TESTER (NAME) values (?)
[java] ID of mypack.IdentityTester:1
[java] ID of mypack.IdentityTester:2
[java] ID of mypack.IdentityTester:3
```

在以上 `insert` 语句中不包含 `ID` 字段，由此可见，Hibernate 在持久化一个 `IdentityTester` 对象时，由底层数据库负责生成主键值。

`identity` 标识符生成器具有以下适用范围。

- 由于 `identity` 生成标识符的机制依赖于底层数据库系统，因此，要求底层数据库系统必须支持自动增长字段类型。支持自动增长字段类型的数据库包括：DB2、MySQL、Ms SQL Server、Sybase、HypersonicSQL、HSQLDB 和 Informix 等。
- OID 必须为 `long`、`int` 或 `short` 类型，如果把 OID 定义为 `byte` 类型，在运行时抛出如下异常：

```
[java] org.hibernate.id.IdentifierGenerationException: this id generator
generates long, integer, short identity
```

#### 4.4.3 sequence 标识符生成器

`sequence` 标识符生成器利用底层数据库提供的序列来生成标识符。例如，在 `SequenceTester.hbm.xml` 中声明使用 `sequence` 标识符生成器：

```
<hibernate-mapping>
```

```
<class name="mypack.SequenceTester" table="SEQUENCE_TESTER">

    <id name="id" type="long" column="ID">
        <meta attribute="scope-set">private</meta>
        <generator class="sequence">
            <param name="sequence">tester_id_seq</param>
        </generator>
    </id>

    <property name="name" type="string" column="NAME" />

</class>
</hibernate-mapping>
```

由于 MySQL 不支持序列，因此本节例子不能用 MySQL 数据库。

### Tips

由于本书例子使用的都是 MySQL，因此在配套光盘中没有提供 SequenceTester.hbm.xml 文件的源代码。

如果 Hibernate 连接的是 Oracle，那么数据库中 SEQUENCE\_TESTER 表的 DDL 定义如下：

```
create table SEQUENCE_TESTER (
    ID bigint not null,
    NAME varchar(15) not null,
    primary key (id)
);
create sequence tester_id_seq;
```

在 DOS 下运行 `ant run_sequence`，以下是在控制台输出的部分信息：

```
[java] insert into SEQUENCE_TESTER (ID,NAME) values (?,?)
[java] insert into SEQUENCE_TESTER (ID,NAME) values (?,?)
[java] insert into SEQUENCE _TESTER (ID,NAME) values (?,?)
[java] ID of mypack.SequenceTester:1
[java] ID of mypack.SequenceTester:2
[java] ID of mypack.SequenceTester:3
```

以上 insert 语句中包含 ID 字段，由此可见，Hibernate 在持久化一个 SequenceTester 对象时，先从底层数据库的 tester\_id\_seq 序列中获得一个唯一的序列号，再把它作为主键值。

sequence 标识符生成器具有以下适用范围。

- 由于 sequence 生成标识符的机制依赖于底层数据库系统的序列，因此，要求底层数据库系统必须支持序列。支持序列的数据库包括：Oracle、DB2、SAP DB 和 PostgreSQL 等。

- OID 必须为 long、int 或 short 类型，如果把 OID 定义为 byte 类型，在运行时抛出如下异常：

```
[java] org.hibernate.id.IdentifierGenerationException: this id generator
generates long, integer, short identity
```

#### 4.4.4 hilo标识符生成器

hilo 标识符生成器由 Hibernate 按照一种 high/low 算法来生成标识符，它从数据库的特定表的字段中获取 high 值。例如，在 HiloTester.hbm.xml 中声明使用 hilo 标识符生成器，其中 high 值存放在 hi\_value 表的 next\_value 字段中：

```
<hibernate-mapping>
  <class name="mypack.HiloTester" table="HILO_TESTER">

    <id name="id" type="long" column="ID">
      <meta attribute="scope-set">private</meta>
      <generator class="hilo">
        <param name="table">hi_value</param>
        <param name="column">next_value</param>
        <param name="max_lo">100</param>
      </generator>
    </id>

    <property name="name" type="string" column="NAME" />

  </class>
</hibernate-mapping>
```

数据库中 HILO\_TESTER 表及 hi\_value 表的 DDL 定义如下：

```
create table HILO_TESTER (
  ID bigint not null,
  name varchar(15),
  primary key (ID)
);
create table hi_value (
  next_value integer
);
insert into hi_value values ( 0 );
```

在 DOS 下运行 ant run\_hilo，以下是在控制台输出的部分信息：

```
[java] insert into HILO_TESTER (ID,NAME) values (?,?)
[java] insert into HILO_TESTER (ID,NAME) values (?,?)
[java] insert into HILO _TESTER (ID,NAME) values (?,?)
[java] ID of mypack.HiloTester:1
[java] ID of mypack. HiloTester:2
```



```
[java] ID of mypack. HiloTester:3
```

以上 insert 语句中包含 ID 字段，由此可见，Hibernate 在持久化一个 HiloTester 对象时，由 Hibernate 负责生成主键值。hilo 标识符生成器在生成标识符时，需要读取并修改 hi\_value 表中的 next\_value 值。这段操作需要在单独的事务中处理。例如在以下代码中，当执行 session.save(object)方法时，hilo 标识符生成器不使用代码中的 Session 对象的当前数据库连接和事务，而是单独在一个新的数据库连接中创建新的事务，然后访问 hi\_value 表，当然，hilo 标识符生成器所执行的操作对应用程序是透明的：

```
tx = session.beginTransaction();
session.save(object);
tx.commit();
```

hilo 标识符生成器具有以下适用范围。

- 由于 hilo 生成标识符的机制不依赖于底层数据库系统，因此适用于所有的数据库系统。
- OID 必须为 long、int 或 short 类型，如果把 OID 定义为 byte 类型，在运行时抛出如下异常：

```
[java] org.hibernate.id.IdentifierGenerationException: this id
generator generates long, integer, short identity
```

- high/low 算法生成的标识符只能在一个数据库中保证唯一。
- 当用户为 Hibernate 自行提供数据库连接，或者 Hibernate 通过 JTA，从应用服务器的数据源获得数据库连接的时候无法使用 hilo，因为这不能保证 hilo 单独在新的数据库连接的事务中访问 hi\_value 表。

#### 4.4.5 native标识符生成器

native 标识符生成器依据底层数据库对自动生成标识符的支持能力，来选择使用 identity、sequence 或 hilo 标识符生成器。native 标识符生成器能自动判断底层数据库提供的生成标识符的机制。例如，如果底层数据库为 MySQL 和 MS SQL Server，就选择 identity 标识符生成器；如果底层数据库为 Oracle，就选择 sequence 标识符生成器。在 NativeTester.hbm.xml 文件中声明使用 native 标识符生成器：

```
<hibernate-mapping>
  <class name="mypack.NativeTester" table="NATIVE_TESTER">

    <id name="id" type="long" column="ID">
      <meta attribute="scope-set">private</meta>
      <generator class="native"/>
    </id>
```

```

        <property name="name" type="string" column="NAME" />

    </class>
</hibernate-mapping>

```

如果 Hibernate 连接的是 MySQL，数据库中的 NATIVE\_TESTER 表的 DDL 定义如下：

```

create table NATIVE_TESTER (
    ID bigint not null auto_increment,
    NAME varchar(15) not null,
    primary key (id)
);

```

如果 Hibernate 连接的是 MS SQL Server，数据库中 NATIVE\_TESTER 表的 DDL 定义如下：

```

create table NATIVE_TESTER (
    ID bigint not null identity,
    NAME varchar(15) not null,
    primary key (id)
);

```

在 DOS 下运行 `ant run_native`，以下是在控制台输出的部分信息：

```

[java] insert into NATIVE_TESTER (NAME) values (?)
[java] insert into NATIVE_TESTER (NAME) values (?)
[java] insert into NATIVE_TESTER (NAME) values (?)
[java] ID of mypack.NativeTester:1
[java] ID of mypack.NativeTester:2
[java] ID of mypack.NativeTester:3

```

在以上 insert 语句中不包含 ID 字段，这是因为当底层数据库为 MySQL 时，其实使用的是 identity 标识符生成器，因此，当 Hibernate 在持久化一个 NativeTester 对象时，由底层数据库负责生成主键值。native 标识符生成器具有以下适用范围。

- 由于 native 能根据底层数据库系统的类型，自动选择合适的标识符生成器，因此很适合于跨数据库平台开发，即同一个 Hibernate 应用需要连接多种数据库系统的场合。
- OID 必须为 long、int 或 short 类型，如果把 OID 定义为 byte 类型，在运行时抛出如下异常：

```

[java] org.hibernate.id.IdentifierGenerationException: this id generator
generates long, integer, short identity

```

## 4.5 映射自然主键

自然主键是具有业务含义的主键。如果从头设计数据库表，应该避免使用自然主键，而尽量使用不具有业务含义的代理主键。对于原有的数据库系统，假如已经使用了自然主键，并且不允许修改关系数据模型，Hibernate 对此也提供了映射方案。下面以 MONKEYS 表为例，介绍映射单个自然主键的方案。

假如 MONKEYS 表没有定义 ID 代理主键，而是以 NAME 字段作为主键，那么相应地，在 Monkey 类中不必定义 id 属性，Monkey 类的 OID 为 name 属性，它的映射代码如下：

```
<class name="mypack.Monkey" table="MONKEYS">
  <id name="name" column="NAME" type="string">
    <generator class="assigned"/>
  </id>

  <version name="version" column="VERSION" unsaved-value="null" />
  .....
</class>
```

在以上代码中，标识符生成策略为“assigned”，表示由应用程序为 name 属性赋值。不管 Monkey 对象是临时对象，还是游离对象，name 属性永远不会为 null，因此 Session 的 saveOrUpdate() 方法无法通过判断 name 属性是否为 null 来确定 Monkey 对象的状态。在这种情况下，可以设置<version>版本控制元素的 unsaved-value 属性。以上代码表明，如果 Monkey 对象的 version 属性为 null，就表示临时对象，否则为游离对象。

与<version>元素对应，在 Monkey 类中有一个 version 属性：

```
private Integer version;

public Integer getVersion() {
    return this.version;
}

private void setVersion(Integer version) {
    this.version = version;
}
```

与<version>元素对应，在 MONKEYS 表中有一个 VERSION 字段：

```
create table MONKEYS (
  NAME varchar(255) not null,
  VERSION integer not null,
  .....
```

```
primary key (NAME));
```

### Tips

本书第 15 章（处理并发问题）将进一步介绍版本控制元素<version>的用法。

以下程序创建了一个 Monkey 对象，然后调用 Session 的 saveOrUpdate() 方法保存它：

```
Monkey monkey=new Monkey();
monkey.setName("Tom");
//由于monkey对象的version属性为null,因此实际上调用save()方法
session.saveOrUpdate(monkey);
System.out.println(session.getIdentifier(monkey));
```

Session 的 getIdentifier() 方法返回 Monkey 对象的 OID，在以上程序中它返回 Monkey 对象的 name 属性，因此以上程序的打印结果为 “Tom”。

## 4.6 小结

关系数据库中的主键可分为自然主键（具有业务含义）和代理主键（不具有业务含义），其中代理主键可以适应不断变化的业务需求，因此更加流行。代理主键通常为整数类型，与此对应，在持久化类中也应该把 OID 定义为整数类型，Hibernate 允许把 OID 定义为 short、int 和 long 类型，以及它们的包装类型。

本章接着介绍了 Hibernate 提供的几种内置标识符生成器的用法。每一种标识符生成器都有它的适用范围，应该根据所使用的数据库和 Hibernate 应用的软件架构来选择合适的标识符生成器。以下总结了几种常用数据库系统可使用的标识符生成器。

- MySQL: identity、increment、hilo、native
- MS SQL Server: identity、increment、hilo、native
- Oracle: sequence、seqhilo、hilo、increment、native
- 跨数据库平台开发: native

本章最后介绍了自然主键的映射方案。对于从头设计的关系数据模型，应该优先考虑使用代理主键。

## 第5章 映射一对多关联关系

在 Java 对象模型中，类与类之间最普遍的关系是关联关系。例如，花果山有许多武术队（用 Team 类表示），一个武术队可以包含多个猴子，而一个猴子（用 Monkey 类表示）最多可以参加一个武术队。因此，Team 类与 Monkey 类之间为一对多的关联关系。

在 UML 语言中，关联是有方向的。从 Monkey 到 Team 的关联是多对一关联，这意味着每个 Monkey 对象都会引用一个 Team 对象，因此在 Monkey 类中应该定义一个 Team 类型的 team 属性，来引用所关联的 Team 对象：

```
private Team team; //表示猴子所属的武术队
public Team getTeam(){
    return team;
}
public void setTeam(Team team){
    this.team=team;
}
```

从 Team 到 Monkey 是一对多关联，这意味着每个 Team 对象会引用一组 Monkey 对象，因此在 Team 类中应该定义一个集合类型的属性，来引用所有关联的 Monkey 对象。

```
private Set monkeys=new HashSet(); //表示武术队中包含的所有猴子
public Set getMonkeys(){
    return monkeys;
}
public void setMonkeys(Set monkeys){
    this.monkeys=monkeys;
}
```

如果仅有从 Monkey 到 Team 的关联（参见图 5-1），或者仅有从 Team 到 Monkey 的关联（参见图 5-2），就称为单向关联。如果同时包含两种关联，就称为双向关联（参见图 5-3）。

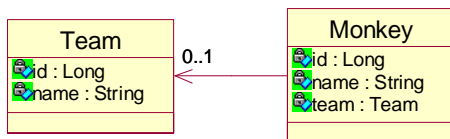


图 5-1 从 Monkey 到 Team 的多对一单向关联

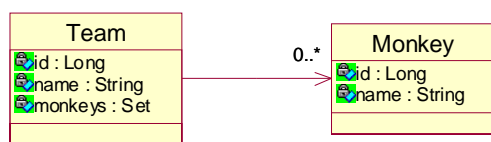


图 5-2 从 Team 到 Monkey 的一对多单向关联

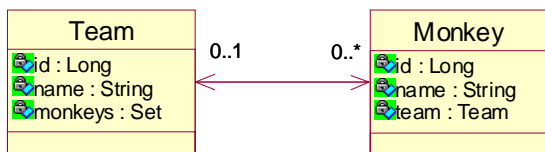


图 5-3 Team 和 Monkey 的一对多双向关联

在关系数据库中，只存在外键参照关系，而且总是由“many”方参照“one”方（参见图 5-4），因为这样才能消除数据冗余，因此关系数据库实际上只支持多对一或一对一的单向关联。

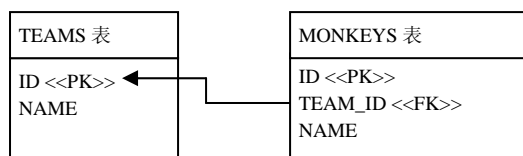


图 5-4 MONKEYS 表参照 TEAMS 表

创建 TEAMS 表和 MONKEYS 表的 DDL 定义如下：

```

create table TEAMS (
    ID bigint not null,
    NAME varchar(15),
    primary key (ID));

create table MONKEYS (
    ID bigint not null,
    NAME varchar(15),
    TEAM_ID bigint,
    primary key (ID));

alter table MONKEYS add index IDX_TEAM(TEAM_ID),
add constraint FK_TEAM foreign key (TEAM_ID) references TEAMS (ID);
    
```

在前面章节，大家已经跟随悟空掌握了把单个 Monkey 类映射到单个 MONKEYS 表的方法。本章将结合具体的例子来介绍如何映射以下关联关系：

- 以 Monkey 和 Team 类为例，介绍如何映射多对一单向关联关系。
- 以 Monkey 和 Team 类为例，介绍如何映射一对多双向关联关系。

## 5.1 建立多对一的单向关联关系

在类与类之间各种各样的关系中，要算多对一的单向关联关系和关系数据库中的外键参照关系最匹配了。因此如果使用单向关联，通常选择从 **Monkey** 到 **Team** 的多对一单向关联。在 **Monkey** 类中需要定义一个 **team** 属性，而在 **Team** 类中无须定义用于存放 **Monkey** 对象的集合属性。例程 5-1 和例程 5-2 分别是 **Team** 类和 **Monkey** 类的源程序。

例程 5-1 Team.java

```
package mypack;
public class Team{
    private Long id;
    private String name;

    //此处省略构造方法，以及 id 和 name 属性的访问方法
    .....
}
```

例程 5-2 Monkey.java

```
package mypack;
public class Monkey{
    private Long id;
    private String name;
    private Team team;

    //此处省略构造方法，以及 id 和 name 属性的访问方法
    .....

    public Team getTeam(){
        return team;
    }
    public void setTeam(Team team) {
        this.team=team;
    }
}
```

**Team** 类的所有属性和 **TEAMS** 表的字段一一对应，因此把 **Team** 类映射到 **TEAMS** 表非常简单，参见例程 5-3。

例程 5-3 Team.hbm.xml

```
<hibernate-mapping >
    <class name="mypack.Team" table="TEAMS" >
        <id name="id" type="long" column="ID">
```

```

        <generator class="increment"/>
    </id>
    <property name="name" type="string" column="NAME" />

</class>
</hibernate-mapping>

```

Monkey 类的 name 属性和 MONKEYS 表的 NAME 字段对应，映射代码如下：

```

<property name="name" type="string" column="NAME" />

```

Monkey 类的 team 属性是 Team 类型，和 MONKEYS 表的外键 TEAM\_ID 对应，那么能否按如下方式映射 team 属性呢？

```

<property name="team" column="TEAM_ID" />

```

在以上映射代码中，team 属性是 Team 类型，而 MONKEYS 表的外键 TEAM\_ID 是整数类型，显然类型不匹配，因此不能使用<property>元素来映射 team 属性，而需要使用<many-to-one>元素：

```

<many-to-one
    name="team"
    column="TEAM_ID"
    class="mypack.Team"
    lazy="false"
/>

```

<many-to-one>元素建立了 team 属性和 MONKEYS 表的外键 TEAM\_ID 之间的映射。它包括以下属性。

- **name**: 设定待映射的持久化类的属性的名字，此处为 Monkey 类的 team 属性。
- **column**: 设定和持久化类的属性对应的表的外键，此处为 MONKEYS 表的外键 TEAM\_ID。
- **class**: 设定待映射的持久化类的属性的类型，此处设定 team 属性为 Team 类型。
- **not-null**: 如果为 true，表示 Monkey 类的 team 属性不允许为 null，该属性的默认值为 false。由于在本例中，允许一个猴子不属于任何武术队，所以 not-null 属性取默认值 null。假如 not-null 属性为 true，那么会改变 Hibernate 的运行时行为，Hibernate 在向数据库中保存 Monkey 对象时，会先检查它的 team 属性是否为 null，如果为 null，就会抛出异常。
- **lazy**: 如果为 proxy，表示对关联的 Team 对象使用延迟检索策略并且使用代理，这是默认值。如果为 false，就意味着当 Hibernate 从数据库中加载 Monkey 对象时，还会立即自动加载与它关联的 Team 对象。本书第 7 章（Hibernate 的检索策略和检索方式）将进一步详细介绍 lazy 属性的用法。



例程 5-4 是 Monkey.hbm.xml 的源代码。

例程 5-4 Monkey.hbm.xml

```
<hibernate-mapping >
  <class name="mypack.Monkey" table="MONKEYS">
    <id name="id" type="long" column="ID">
      <generator class="increment"/>
    </id>
    <property name="name" type="string" column="NAME" />

    <many-to-one
      name="team"
      column="TEAM_ID"
      class="mypack.Team"
      lazy="false"
    />
  </class>
</hibernate-mapping>
```

本节的范例程序位于配套光盘的 sourcecode\chapter5\5.1 目录下。在 chapter5 目录下有 2 个 ANT 的工程文件，分别为 build1.xml 和 build2.xml，它们的区别在于文件开头设置的路径不一样，例如在 build1.xml 文件中设置了以下路径：

```
<property name="source.root" value="5.1/src"/>
<property name="class.root" value="5.1/classes"/>
<property name="lib.dir" value="lib"/>
<property name="schema.dir" value="5.1/schema"/>
```

在 DOS 命令行下进入 chapter5 根目录，然后输入命令：

```
ant -file build1.xml run
```

ANT 命令的 -file 选项用于显式指定工程文件。该命令将运行 BusinessService 类，它的源程序参见例程 5-5。

例程 5-5 BusinessService 类

```
package mypack;

import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class BusinessService{
  public static SessionFactory sessionFactory;

  /** 初始化 Hibernate, 创建 SessionFactory 实例 */
  static{.....}
```

```

    /**查询与参数指定的 Team 对象关联的所有 Monkey 对象
    public List findMonkeysByTeam(Team team) {.....}

    /** 按照参数指定的 OID 查询 Team 对象 */
    public Team findTeam(long team_id) {.....}

    /**级联保存 Monkey 和 Team 对象 */
    public void saveTeamAndMonkeyWithCascade(){..... }

    /**分别保存 Team 和 Monkey 对象 */
    public void saveTeamAndMonkey(){.....}

    /**打印 Monkey 对象信息 */
    public void printMonkeys(List monkeys){
        for (Iterator it = monkeys.iterator(); it.hasNext();) {
            Monkey monkey=(Monkey)it.next();
            System.out.println("Monkeys in "+monkey.getTeam().getName()
                +" :"+monkey.getName());
        }
    }

    public void test(){
        saveTeamAndMonkey();
        saveTeamAndMonkeyWithCascade();
        Team team=findTeam(1);
        List monkeys=findMonkeysByTeam(team);
        printMonkeys(monkeys);
    }

    public static void main(String args[]){
        new BusinessService().test();
        sessionFactory.close();
    }
}

```

BusinessService 类的 main()方法调用 test()方法, test()方法又依次调用以下方法。

(1) saveTeamAndMonkey(): 先创建并持久化一个 Team 对象, 然后创建两个 Monkey 对象, 它们都和这个 Team 对象关联, 最后持久化这两个 Monkey 对象:

```

tx = session.beginTransaction();
Team team=new Team("DREAM");
session.save(team);

Monkey monkey1=new Monkey("Jack",team);
Monkey monkey2=new Monkey("Bill",team);
session.save(monkey1);

```

```
session.save(monkey2);
tx.commit();
```

运行 `saveTeamAndMonkey()` 方法时, Hibernate 将执行以下 3 条 insert 语句:

```
insert into TEAMS (ID,NAME) values (1, 'DREAM');
insert into MONKEYS (ID,NAME, TEAM_ID) values (1, 'Jack', 1);
insert into MONKEYS (ID,NAME, TEAM_ID) values (2, 'Bill', 1);
```

(2) `saveTeamAndMonkeyWithCascade()`: 该方法 and `saveTeamAndMonkey()` 方法只有一个区别, 前者没有先调用 `session.save(team)` 方法持久化 Team 对象, 而是仅持久化了两个 Monkey 对象:

```
tx = session.beginTransaction();
Team team=new Team("BULL");
//session.save(team); 不保存 team 对象

Monkey monkey1=new Monkey("Tom",team);
Monkey monkey2=new Monkey("Mike",team);

session.save(monkey1);
session.save(monkey2);

tx.commit();
```

执行该方法时会抛出异常, 5.1.1 节会详细分析出现异常的原因。

(3) `findTeam()`: 按照参数指定的 OID 来查询 Team 对象。

(4) `findMonkeysByTeam()`: 按照参数指定的 Team 对象来查询相关的 Monkey 对象, 它使用了如下 HQL 查询语句:

```
List monkeys=
    session.createQuery("from Monkey as m where m.team.id="
        +team.getId()).list();
```

运行 `findMonkeysByTeam()` 方法时, Hibernate 将执行以下 select 语句:

```
select * from MONKEYS where TEAM_ID=1;
```

(5) `printMonkeys()`: 打印参数指定的 monkeys 集合中所有的 Monkey 对象及所关联的 Team 对象的信息。由于 Monkey 和 Team 之间存在多对一的单向关联关系, 因此只要调用 `monkey.getTeam()` 方法, 就可以方便地从 Monkey 对象导航到 Team 对象:

```
System.out.println("Monkeys in "+monkey.getTeam().getName()+
    ":" +monkey.getName());
```

`printMonkeys()` 方法在 DOS 控制台输出的结果为:

```
Monkeys in DREAM :Jack
```

Monkeys in DREAM :Bill

### 5.1.1 关于TransientObjectException异常

当执行 `saveTeamAndMonkeyWithCascade()` 方法时，会抛出如下异常：

```
[java] org.hibernate.TransientObjectException: object references an unsaved
transient instance - save the transient instance before flushing: mypack.Team
```

这是在执行 `tx.commit()` 方法时抛出的异常：

```
tx = session.beginTransaction();
Team team=new Team("BULL");

Monkey monkey1=new Monkey("Tom",team);
Monkey monkey2=new Monkey("Mike",team);

session.save(monkey1);
session.save(monkey2);

tx.commit(); //抛出 TransientObjectException 异常
```

下面分析产生异常的原因。在调用 `session.save(monkey1)` 和 `session.save(monkey2)` 方法之前，`monkey1`、`monkey2` 和 `team` 对象都是临时（transient）对象。临时对象是指刚通过 `new` 语句创建，并且还没有被持久化的对象。

当 `session.save(monkey1)` 方法执行完毕，`monkey1` 对象被成功持久化，就变成了持久化对象，而 Hibernate 不会自动持久化 `monkey1` 所关联的 `team` 对象，这意味着 `session.save(monkey1)` 方法仅向数据库中的 MONKEYS 表中插入了一条记录，并且该记录的 TEAM\_ID 字段为 NULL。

#### Tips

如果要更加详细地了解持久化对象和临时对象的区别，参见本书第 6 章（通过 Hibernate 操纵对象）。

同理，当 `session.save(monkey2)` 方法执行成功，`monkey2` 对象被成功持久化，就变成了持久化对象，而与 `monkey2` 所关联的 `team` 对象仍然是临时对象。

`monkey1` 和 `monkey2` 对象作为持久化对象，都位于 Session 的缓存中。当 Hibernate 执行 `tx.commit()` 方法，试图提交事务之前，会先清理 Session 缓存中的所有持久化对象，此时抛出了 `TransientObjectException` 异常。

所谓清理（flush），是指 Hibernate 按照持久化对象的属性变化来同步更新数据库。Hibernate 发现持久化对象 `monkey1` 和 `monkey2` 都引用临时对象 `team`，而在 MONKEYS 表中相应的两条记录的 TEAM\_ID 字段为 NULL，这意味着内存中的持久化对象的属性和数据库中记录不一致，参见图 5-5，而且在这种情况下，Hibernate

没办法使两者同步，因为 Hibernate 不会自动持久化 team 对象，所以 Hibernate 会抛出 `TransientObjectException` 异常，错误原因是持久化对象 monkey1 和 monkey2 的 team 属性引用了一个临时对象 team。

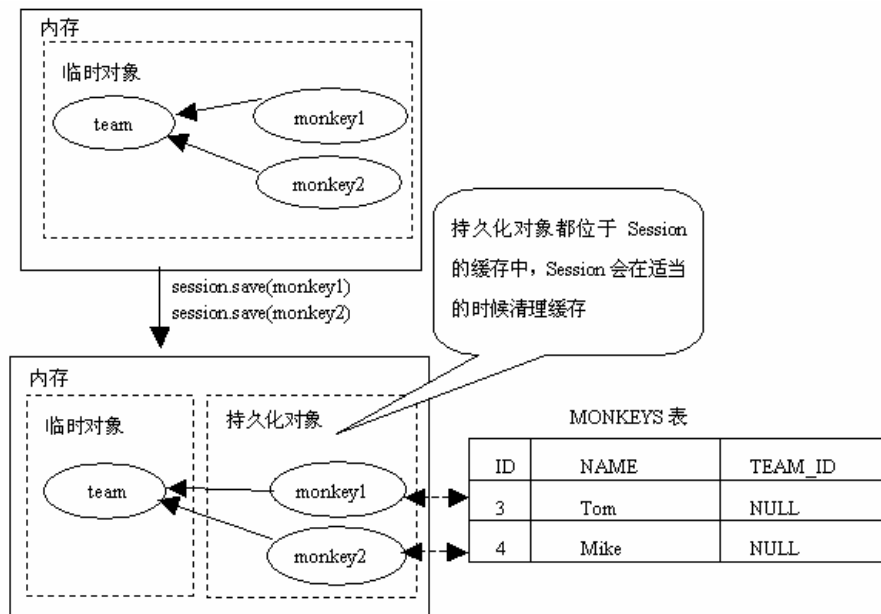


图 5-5 持久化对象 monkey1 和 monkey2 引用临时对象 team

### 5.1.2 级联保存和更新

根据 5.1.1 节可以看出，当 Hibernate 持久化一个临时对象时，在默认情况下，它不会自动持久化所关联的其他临时对象，所以会抛出 `TransientObjectException` 异常。如果希望当 Hibernate 持久化 Monkey 对象时自动持久化所关联的 Team 对象，可以把 `<many-to-one>` 的 `cascade` 属性设为 “`save-update`”，`cascade` 属性的默认值为 “`none`”：

```
<many-to-one
  name="team"
  column="TEAM_ID"
  class="mypack.Team"
  cascade="save-update"
  lazy="false"
/>
```

再执行 `saveTeamAndMonkeyWithCascade()` 方法中的 `session.save(monkey1)` 方法时，Hibernate 把 monkey1 和 team 对象一起持久化，此时 Hibernate 执行的 SQL 语句如下：

```
insert into TEAMS (ID,NAME) values (2, "BULL")
```

```
insert into MONKEYS(ID,NAME,TEAM_ID)values(3,"Tom",2)
```

当 `cascade` 属性为 “`save-update`”，表明保存或更新当前对象时（即执行 `insert` 或 `update` 语句时），会级联保存或更新与它关联的对象。

## 5.2 映射一对多双向关联关系

当类与类之间建立了关联，就可以方便地从一个对象导航到另一个或者一组与它关联的对象。例如对于给定的 `Monkey` 对象，如果想获得与它关联的 `Team` 对象，只要调用如下方法：

```
Team team=monkey.getTeam();//从 Monkey 对象导航到关联的 Team 对象
```

那么对于给定的 `Team` 对象，如果想获得与它关联的所有 `Monkey` 对象，该如何处理呢？在 5.1 节中，由于 `Team` 对象不和 `Monkey` 对象关联，因此必须通过 Hibernate API 查询数据库：

```
List monkeys=
    session.createQuery("from Monkey as m where m.team.id="
        +team.getId()).list();
```

对象位于内存中，在内存中从一个对象导航到另一个对象显然比到数据库中查询数据的速度快多了。但是复杂的关联关系也会给编程带来麻烦，随意修改一个对象，就有可能牵一发而动全身，必须调整许多与之关联的对象之间的关系。类与类之间到底是建立单向关联，还是双向关联，这是由业务需求决定的。以 `Team` 类和 `Monkey` 类为例，如果软件应用有大量这样的需求：

- 根据给定的武术队，查询该武术队的所有猴子。
- 根据给定的猴子，查询所属的武术队。

根据以上需求，不妨为 `Team` 类和 `Monkey` 类建立一对多双向关联。在 5.1 节的例子中，已经建立了 `Monkey` 类到 `Team` 类的多对一关联，下面再增加 `Team` 类到 `Monkey` 类的一对多关联，这需要在 `Team` 类中增加一个集合类型的 `monkeys` 属性：

```
private Set monkeys=new HashSet();
public Set getMonkeys(){
    return monkeys;
}
public void setMonkeys(Set monkeys){
    this.monkeys=monkeys;
}
```

### Tips

既然是双向关联，“一对多双向关联”和“多对一双向关联”是同一回事，只不过“一对多双向关联”听起来更顺口。

有了以上属性，对于给定的武术队，查询该武术队的所有猴子，只需要调用 `team.getMonkeys()` 方法。Hibernate 要求在持久化类中定义集合类属性时，必须把属性声明为接口类型，如 `java.util.Set`、`java.util.Map` 和 `java.util.List`，关于这几个接口的用法，参见本书第 11 章（Java 集合类）。声明为接口可以提高持久化类的透明性，当 Hibernate 调用 `setMonkeys(Set monkeys)` 方法时，传递的参数是 Hibernate 自定义的实现该接口的类的实例。如果把 `monkeys` 声明为 `java.util.HashSet` 类型（它是 `java.util.Set` 接口的一个实现类），就强迫 Hibernate 只能把 `HashSet` 类的实例传给 `setMonkeys()` 方法，本书第 12 章的 12.6 节（小结）对此做了进一步解释。

在定义 `monkeys` 集合属性时，通常把它初始化为集合实现类的一个实例，例如：

```
private Set monkeys=new HashSet();
```

这可以提高程序的健壮性，避免应用程序访问取值为 `null` 的 `monkeys` 集合的方法而抛出 `NullPointerException`。例如以下程序访问 `Team` 对象的 `monkeys` 集合，即使 `monkeys` 集合中不包含任何元素，但是调用 `monkeys.iterator()` 方法不会抛出 `NullPointerException` 异常，因为 `monkeys` 集合并不为 `null`：

```
Set monkeys=team.getMonkeys();
Iterator it=monkeys.iterator();
while(it.hasNext()){
    .....
}
```

例程 5-6 是本节的 `Team.java` 的源程序。

例程 5-6 Team.java

```
package mypack;
import java.util.Set;
import java.util.HashSet;
public class Team{
    private Long id;
    private String name;
    private Set monkeys=new HashSet();

    //此处省略构造方法，以及 id 和 name 属性的访问方法
    .....
    public Set getMonkeys(){
        return monkeys;
    }
    public void setMonkeys(Set monkeys) {
        this.monkeys=monkeys;
    }
}
```

接下来的问题是如何在映射文件中映射集合类型的 `monkeys` 属性，由于在

TEAMS 表中没有直接与 `monkeys` 属性对应的字段，因此不能用 `<property>` 元素来映射 `monkeys` 属性，而是要使用 `<set>` 元素：

```
<set
    name="monkeys"
    cascade="save-update"
    >

    <key column="TEAM_ID" />
    <one-to-many class="mypack.Monkey" />
</set>
```

`<set>` 元素包括以下属性。

- `name`：设定待映射的持久化类的属性名，这里为 `Team` 类的 `monkeys` 属性。
- `cascade`：当取值为 “`save-update`”，表示级联保存和更新。

`<set>` 元素还包含两个子元素：`<key>` 和 `<one-to-many>`。`<one-to-many>` 元素设定所关联的持久化类，此处为 `Monkey` 类，`<key>` 元素设定与所关联的持久化类对应的表的外键，此处为 `MONKEYS` 表的 `TEAM_ID` 字段。

Hibernate 根据以上映射代码获得以下信息。

- `<set>` 元素表明 `Team` 类的 `monkeys` 属性为 `java.util.Set` 集合类型。
- `<one-to-many>` 子元素表明 `monkeys` 集合中存放的是一组 `Monkey` 对象。
- `<key>` 子元素表明 `MONKEYS` 表通过外键 `TEAM_ID` 参照 `TEAMS` 表。
- `cascade` 属性取值为 “`save-update`”，表明当保存或更新 `Team` 对象时，会级联保存或更新 `monkeys` 集合中的所有 `Monkey` 对象。

例程 5-7 是 `Team.hbm.xml` 的源代码。`Monkey.hbm.xml` 的源代码和 5.1 节的例程 5-4 相同。

例程 5-7 Team.hbm.xml

```
<hibernate-mapping >

    <class name="mypack.Team" table="TEAMS" >
        <id name="id" type="long" column="ID">
            <generator class="increment" />
        </id>

        <property name="name" type="string" column="NAME" />

        <set
            name="monkeys"
            cascade="save-update"
            >
```



```

        <key column="TEAM_ID" />
        <one-to-many class="mypack.Monkey" />
    </set>

</class>
</hibernate-mapping>

```

本节的范例程序位于配套光盘的 sourcecode\chapter5\5.2 目录下。在 DOS 命令行下进入 chapter5 根目录，然后输入命令：

```
ant -file build2.xml run
```

该命令将运行 BusinessService 类，它的源程序参见例程 5-8。

#### 例程 5-8 BusinessService.java

```

package mypack;

import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class BusinessService{
    public static SessionFactory sessionFactory;
    private Long idOfTom;
    private Long idOfBULL;
    private Long idOfJack;
    private Long idOfDREAM;

    /** 初始化 Hibernate，创建 SessionFactory 实例 */
    static{.....}

    /** 打印 Team 对象的所有 Monkey 对象 */
    public void printMonkeysOfTeam(Long teamId){..... }

    /** 级联保存 Team 和 Monkey 对象 */
    public void saveTeamAndMonkeyWithCascade(){..... }

    /** 建立 Team 和 Monkey 对象的关联关系 */
    public void associateTeamAndMonkey(){..... }

    /** 分别保存 Team 和 Monkey 对象 */
    public void saveTeamAndMonkeySeparately(){..... }

    /** 删除一个 Team 对象 */
    public void deleteTeam(Long teamId) {..... }

```

```

    /** 解除一个 Monkey 对象和 Team 对象的关联关系 */
    public void removeMonkeyFromTeam(Long teamId) {..... }

    /**打印 Monkey 对象的信息 */
    public void printMonkeys(Set monkeys){.....}

    public void saveTeamAndMonkeyWithInverse(){
        saveTeamAndMonkeySeparately();
        associateTeamAndMonkey();
    }

    public void test(){
        saveTeamAndMonkeyWithCascade();
        saveTeamAndMonkeyWithInverse();
        printMonkeysOfTeam(idOfBULL);
        deleteTeam(idOfDREAM);
        removeMonkeyFromTeam(idBULL);
    }

    public static void main(String args[]){
        new BusinessService().test();
        sessionFactory.close();
    }
}

```

BusinessService 类的 main()方法调用 test()方法, test()方法又依次调用以下方法。

(1) saveTeamAndMonkeyWithCascade(): 该方法用于演示当 Team.hbm.xml 文件中的<set>元素的 cascade 属性为“save-update”时 Hibernate 的运行时行为。该方法先创建一个 Team 对象和 Monkey 对象, 接着建立两者的一对多双向关联关系, 最后调用 session.save(team)方法持久化 Team 对象:

```

tx = session.beginTransaction();
//创建一个 Team 对象和 Monkey 对象
Team team=new Team("BULL",new HashSet());
Monkey monkey=new Monkey();
monkey.setName("Tom");

//建立 Team 对象和 Monkey 对象的一对多双向关联关系
monkey.setTeam(team);
team.getMonkeys().add(monkey);

//保存 Team 对象
session.save(team);

tx.commit();

```

当<set>元素的 cascade 属性为“save-update”时，Hibernate 在持久化 Team 对象时，会自动持久化关联的所有 Monkey 对象。Hibernate 将执行以下两条 insert 语句：

```
insert into TEAMS (ID,NAME) values (1, "BULL")
insert into MONKEYS (ID,NAME,TEAM_ID) values
(1,"Tom",1)
```

(2) saveTeamAndMonkeyWithInverse(): 该方法用于演示<set>元素的 inverse 属性的用法，5.2.1 节会对此详细介绍。

(3) printMonkeysOfTeam(): 打印与 Team 对象关联的 Monkey 对象。该方法先加载 Team 对象，接下来调用 team.getMonkeys()方法，就能在内存中从 Team 对象导航到所有关联的 Monkey 对象。

```
tx = session.beginTransaction();
Team team=
    (Team)session.get(Team.class,new Long(teamId));
printMonkeys(team.getMonkeys());
tx.commit();
```

(4) deleteTeam(): 该方法用于演示当<set>元素的 cascade 属性取值为“delete”时，Hibernate 的运行时行为。5.2.2 节会对此详细介绍。

(5) removeMonkeyFromTeam(): 该方法用于演示当<set>元素的 cascade 属性取值为“all-delete-orphan”时，Hibernate 的运行时行为。5.2.3 节会对此详细介绍。

### 5.2.1 <set>元素的inverse属性

saveTeamAndMonkeyWithInverse()方法用于演示<set>元素的 inverse 属性的用法。该方法依次调用以下两个方法。

(1) saveTeamAndMonkeySeparately()方法：先创建一个 Team 对象和一个 Monkey 对象，不建立它们的关联关系，最后分别持久化这两个对象：

```
tx = session.beginTransaction();

Team team=new Team();
team.setName("DREAM");

Monkey monkey=new Monkey();
monkey.setName("Jack");

session.save(team);
session.save(monkey);

tx.commit();
```

Hibernate 将执行以下两条 insert 语句：

```
insert into TEAMS (ID,NAME) values (2, "DREAM");
insert into MONKEYS (ID,NAME,TEAM_ID) values (2, "Jack",NULL);
```

(2) `associateTeamAndMonkey()`：该方法加载被 `saveTeamAndMonkeySeparately()` 方法持久化的 `Team` 和 `Monkey` 对象，然后建立两者的一对多双向关联关系：

```
tx = session.beginTransaction();

//加载持久化对象 Team 和 Monkey
Team team=(Team)session.load(Team.class,idOfDREAM);
Monkey monkey=(Monkey)session.load(Monkey.class,idOfJack);

//建立 Team 和 Monkey 的关联关系
monkey.setTeam(team);
team.getMonkeys().add(monkey);

tx.commit();
```

Hibernate 会自动清理缓存中的所有持久化对象，按照持久化对象的属性变化来同步更新数据库。如果把 `Team.hbm.xml` 文件中 `<set>` 元素的 `inverse` 属性设为 `false`，那么 Hibernate 在清理以上 `Team` 对象和 `Monkey` 对象时执行以下两条 SQL 语句。

```
update MONKEYS set NAME='Jack', TEAM_ID=2 where ID=2;
update MONKEYS set TEAM_ID=2 where ID=2;
```

尽管实际上只是修改了 `MONKEYS` 表的一条记录，但是以上 SQL 语句表明 Hibernate 执行了两次 `update` 操作。这是因为 Hibernate 根据内存中持久化对象的属性变化来决定需要执行哪些 SQL 语句。当建立 `monkey` 对象和 `team` 对象的双向关联关系时，需要在程序中分别修改这两个对象的属性：

(1) 修改 `Monkey` 对象，建立 `Monkey` 对象到 `Team` 对象的多对一关联关系：

```
monkey.setTeam(team);
```

Hibernate 探测到持久化对象 `Monkey` 的属性的上述变化后，执行相应的 SQL 语句为：

```
update MONKEYS set NAME='Jack', TEAM_ID=2 where ID=2;
```

(2) 修改 `Team` 对象，建立 `Team` 对象到 `Monkey` 对象的一对多关联关系：

```
team.getMonkeys().addMonkey(monkey);
```

Hibernate 探测到持久化对象 `Team` 的属性的上述变化后，执行相应的 SQL 语句为：

```
update MONKEYS set TEAM_ID=2 where ID=2;
```

重复执行多余的 SQL 语句会影响 Java 应用的性能, 解决这一问题的办法是把 <set> 元素的 inverse 属性设为 true, 该属性的默认值为 false:

```
<set
    name="monkeys"
    cascade="save-update"
    inverse="true" >

    <key column="TEAM_ID" />
    <one-to-many class="mypack.Monkey" />
</set>
```

以上代码表明在 Team 和 Monkey 的双向关联关系中, Team 端的关联只是 Monkey 端关联的镜像。当 Hibernate 探测到持久化对象 Team 和 Monkey 的属性均发生变化时, 仅按照 Monkey 对象属性的变化来同步更新数据库。

按照上述方式修改 Team.hbm.xml, 再运行 associateTeamAndMonkey() 方法, Hibernate 仅执行一条 SQL 语句:

```
update MONKEYS set NAME='Jack', TEAM_ID=2 where ID=2;
```

如果对 associateTeamAndMonkey() 方法做如下修改, 粗体字为修改部分:

```
tx = session.beginTransaction();

//加载持久化对象 Team 和 Monkey
Team team=(Team)session.load(Team.class,idOfDREAM);
Monkey monkey=(Monkey)session.load(Monkey.class,idOfJack);

monkey.setTeam(team); //建立 Monkey 到 Team 的关联关系
//team.getMonkeys().add(monkey); 不建立 Team 到 Monkey 的关联

tx.commit();
```

以上代码仅设置了 Monkey 对象的 team 属性, Hibernate 仍然会按照 Monkey 对象的属性的变化来同步更新数据库, 执行以下 SQL 语句:

```
update MONKEYS set NAME='Jack', TEAM_ID=2 where ID=2;
```

如果对 associateTeamAndMonkey() 方法做如下修改, 粗体字为修改部分:

```
tx = session.beginTransaction();

//加载持久化对象 Team 和 Monkey
Team team=(Team)session.load(Team.class,idOfDREAM);
Monkey monkey=(Monkey)session.load(Monkey.class,idOfJack);

//monkey.setTeam(team); 不建立 Monkey 到 Team 的关联
team.getMonkeys().add(monkey); //建立 Team 到 Monkey 的关联关系
```

```
tx.commit();
```

以上代码仅设置了 Team 对象的 monkeys 属性，由于<set>元素的 inverse 属性为 true，因此，Hibernate 不会按照 Team 对象的属性变化来同步更新数据库。

根据上述实验，可以得出这样的结论。

- 在映射一对多的双向关联关系时，应该在“one”方把<set>元素的 inverse 属性设为 true，这可以提高应用的性能。
- 在建立两个对象的双向关联时，应该同时修改关联两端的对象的相应属性：

```
monkey.setTeam(team);
team.getMonkeys().add(monkey);
```

这样才会使程序更加健壮，提高业务逻辑层的独立性，使业务逻辑层的程序代码不受 Hibernate 实现的影响。同理，当解除双向关联的关系时，也应该修改关联两端的对象的相应属性：

```
monkey.setTeam(null);
team.getMonkeys().remove(monkey);
```

### 5.2.2 级联删除

在 deleteTeam()方法中，先加载一个 Team 对象，然后删除这个对象：

```
tx = session.beginTransaction();
Team team=(Team)session.load(Team.class,teamId);
session.delete(team);
tx.commit();
```

如果 cascade 属性取默认值“none”，当 Hibernate 删除一个持久化对象时，不会自动删除与它关联的其他持久化对象。如果希望 Hibernate 删除 Team 对象时，自动删除和 Team 关联的 Monkey 对象，可以把 cascade 属性设为“delete”：

```
<set
    name="monkeys"
    cascade="delete"
    inverse="true"
>
    <key column="TEAM_ID" />
    <one-to-many class="mypack.Monkey" />
</set>
```

再运行 deleteTeam()方法时，Hibernate 会同时删除 Team 对象及关联的 Monkey 对象，此时 Hibernate 执行以下 SQL 语句：

```
delete from MONKEYS where TEAM_ID=2;
```

```
delete from TEAMS where ID=2;
```

### Tips

所谓删除一个持久化对象，并不是指从内存中删除这个对象，而是指从数据库中删除相关的记录。这个对象依然存在于内存中，只不过由持久化状态转变为删除状态。

## 5.2.3 父子关系

`removeMonkeyFromTeam()`方法先加载一个 `Team` 对象，然后获得与 `Team` 对象关联的一个 `Monkey` 对象的引用，最后解除 `Team` 和 `Monkey` 对象之间的关系：

```
tx = session.beginTransaction();

//加载 Team 对象
Team team=(Team)session.load(Team.class,teamId);

//获得与 Team 对象关联的一个 Monkey 对象的引用
Monkey monkey=(Monkey)team.getMonkeys().iterator().next();

//解除 Team 对象和 Monkey 对象的关联关系
team.getMonkeys().remove(monkey);
monkey.setTeam(null);

tx.commit();
```

如果 `cascade` 属性取默认值“none”，当 Hibernate 解除 `Team` 和 `Monkey` 对象之间的关联关系时，会执行以下 SQL 语句，使得 `MONKEYS` 表中的相应记录不再参照 `TEAMS` 表：

```
update MONKEYS set NAME='Tom',TEAM_ID=NULL where ID=1;
```

如果希望 Hibernate 自动删除不再和 `Team` 对象关联的 `Monkey` 对象，可以把 `cascade` 属性设为“all-delete-orphan”：

```
<set
    name="monkeys"
    cascade="all-delete-orphan"
    inverse="true"
  >
    <key column="TEAM_ID" />
    <one-to-many class="mypack.Monkey" />
  </set>
```

再运行 `removeMonkeyFromTeam()`方法时，Hibernate 会执行以下 SQL 语句：

```
delete from MONKEYS where TEAM_ID=1 and ID=1;
```

当 `Team.hbm.xml` 的 `<set>` 元素的 `cascade` 属性取值为“all-delete-orphan”，它将

包含“all”和“delete-orphan”的行为。关于 cascade 属性的所有可选值的用法可参考第 6 章的 6.4 节的表 6-3（cascade 属性）。

当关联双方存在父子关系，就可以把父方的 cascade 属性设为“all-delete-orphan”。所谓父子关系，是指由父方来控制子方的持久化生命周期，子方对象必须和一个父方对象关联，而不允许单独存在。如果删除父方对象，应该级联删除所有关联的子方对象；如果一个子方对象不再和一个父方对象关联，也应该把这个子方对象删除。

类与类之间是否存在父子关系是由业务需求决定的。通常认为武术队（Team）和猴子（Monkey）之间不存在父子关联关系，一个猴子可以不属于任何一个武术队。此外，当一个猴子离开一个武术队后，也可以加入到另一个武术队。以下代码演示猴子 Tom 离开 BULL 武术队，加入到 DREAM 武术队：

```
//解除猴子 Tom 与 BULL 武术队的关联关系
teamBULL.getMonkeys().remove(monkeyTom);
monkeyTom.setTeam(null);

//建立猴子 Tom 与 DREAM 武术队的关联关系
teamDREAM.getMonkeys().add(monkeyTom);
monkeyTom.setTeam(teamDREAM);
```

通常认为客户（Customer）和订单（Order）之间存在父子关联关系，订单总是由某个客户发出的，因此一条不属于任何客户的订单是没有存在意义的。所以当删除一个 Customer 对象时，需要自动删除与它关联的所有 Order 对象，此外，如果解除了 Order 对象与 Customer 对象的关联关系，也应该删除这个 Order 对象。

## 5.3 小结

现在，悟空已经能熟练地把一个 Team 对象及关联的若干 Monkey 对象同时保存到数据库中，在需要的时候，还能把它们一起加载到内存中。

本章介绍了一对多关联关系的映射方法，重点介绍了 inverse 属性和 cascade 属性的用法。在映射双向关联关系时，应该在“one”方把<set>元素的 inverse 属性设为 true。cascade 属性用来控制级联操作，可选值包括 save-update、delete 和 all-delete-orphan 等，默认值为 none。第 6 章的 6.4 节的表 6-3 归纳了 cascade 属性的用法。

本章还介绍了通过 Hibernate API 来保存、修改和删除具有关联关系的对象的方法。当 inverse 和 cascade 属性取不同值时，Hibernate 有着不同的运行时行为。本章涉及了好几个新概念：临时对象、持久化对象和清理（flush），第 6 章（通过 Hibernate 操纵对象）会对此做进一步解释。



## 第6章 通过 Hibernate 操纵对象

Session 接口是 Hibernate 向应用程序提供的操纵数据库的最主要接口。现在，悟空已经掌握了通过 Session 接口来进行保存、更新、删除和加载 Java 对象的基本方法。不过，悟空还需要对 Session 接口有更深入的理解，否则就会编写出一些似是而非的代码，这样的代码要么是错误的，要么虽然可以运行但会影响运行性能。例如：

```
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(monkey);
monkey.setName("Linda")
session.save(monkey); //这步操作是多余的
tx.commit();
session.close();
session.save(monkey);
//这步操作是错误的，Session 关闭后就不能再用它来保存对象
```

要深入理解 Session 的运行机制，就必须了解 Session 的缓存。位于 Session 缓存中的对象称为持久化对象，它和数据库中的相关记录对应，Session 能够在某些时间点，按照缓存中对象的变化来执行相关 SQL 语句，来同步更新数据库，这一过程被称为清理缓存（flush）。

站在持久化层的角度，Hibernate 把对象分为 4 种状态：持久化状态、临时状态、游离状态和删除状态。Session 的特定方法能使对象从一个状态转换到另一个状态。

本章先介绍 Session 的缓存，然后介绍对象的 4 种状态及状态转换条件，接着介绍 Session 接口的主要方法，级联操纵对象图的方法，以及批量处理数据的方法。

### 6.1 理解 Session 的缓存

如果希望一个 Java 对象 A 一直处于生命周期中，就必须保证至少有一个变量引用它，或者可以从其他处于生命周期中的对象 B 导航到这个对象 A，比如在对象 B 的 Java 集合属性中存放了对象 A 的引用。在 Session 接口的实现中包含一系列的 Java 集合，这些 Java 集合构成了 Session 的缓存。如图 6-1 所示，只要 Session 实例没有结束生命周期，存放在它缓存中的对象也不会结束生命周期。

当 Session 的 save()方法持久化一个 Monkey 对象时，Monkey 对象被加入到 Session 的缓存中，以后即使应用程序中的引用变量不再引用 Monkey 对象，只要 Session 的缓存还没有被清空，Monkey 对象仍然处于生命周期中。

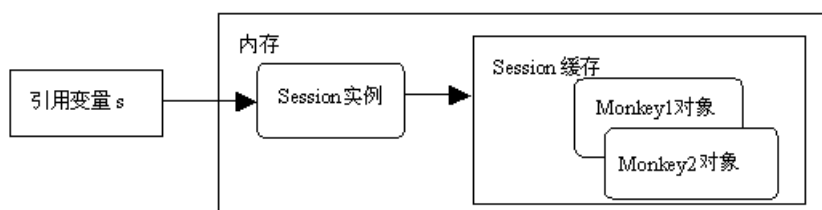


图 6-1 Session 缓存中对象的生命周期依赖于 Session 实例

当 Session 的 `get()` 方法试图从数据库中加载一个 Monkey 对象时, Session 先判断缓存中是否已经存在这个 Monkey 对象, 如果存在, 就不需要再到数据库中检索, 而直接从缓存中获得这个 Monkey 对象。

在以下例程 6-1 的程序代码中, 当调用 Session 的 `save()` 方法持久化 Monkey 对象时, Monkey 对象被加入到 Session 的缓存中。接下来把引用变量 `m1` 置为 `null`, 但是 Monkey 对象仍然位于 Session 的缓存中, 因此它仍然处于生命周期中。当调用 Session 的 `get()` 方法再加载该对象时, 只需从缓存中读取 Monkey 对象, 而不需要通过 SQL `select` 语句到数据库中重新加载。

#### 例程 6-1 Session 的缓存与 Monkey 对象

```

Transaction tx = session.beginTransaction();
Monkey m1=new Monkey("Tom");

//Monkey 对象被持久化, 并且加入到 Session 的缓存中
session.save(m1);
Long id=m1.getId();

//m1 变量不再引用 Monkey 对象
m1=null;
//从 Session 缓存中读取 Monkey 对象, 使 m2 变量引用 Monkey 对象
Monkey m2=(Monkey) session.get(Monkey.class,id);
tx.commit();
//关闭 Session, 清空缓存, Monkey 对象不再位于 Session 缓存中
session.close();

//访问 Monkey 对象
System.out.println(m2.getName());

//m2 变量不再引用 Monkey 对象, 此时 Monkey 对象结束生命周期
m2=null;

```

对于以上程序代码, 当调用 `session.close()` 方法时, Session 的缓存被清空, 但是由于引用变量 `m2` 仍然引用 Monkey 对象, 所以 Monkey 对象依然处于生命周期中。当程序代码最后把 `m2` 变量置为 `null`, 此时 Monkey 对象不再被任何变量或缓存引用, 这时它才结束生命周期。图 6-2 显示了先后引用 Monkey 对象的变量 `m1`、Session 的缓存和变量 `m2`。

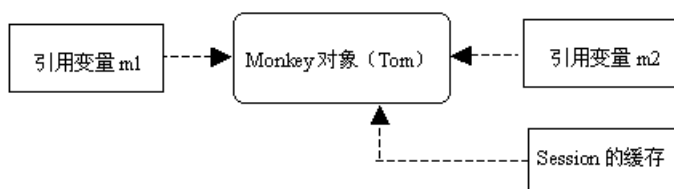


图 6-2 Monkey 对象先后被变量 m1、Session 的缓存和变量 m2 引用

### 6.1.1 Session 的缓存的作用

Session 的缓存有三大作用。

(1) 减少访问数据库的频率。应用程序从缓存中读取持久化对象的速度显然比到数据库中查询数据的速度快多了，因此 Session 的缓存可以提高数据访问的性能。

例如以下代码试图两次加载同一个 Monkey 对象：

```

Transaction tx = session.beginTransaction();
//第一次执行 Session 的 get() 方法
Monkey m1=(Monkey) session.get(Monkey.class,new Long(1));
//第二次执行 Session 的 get() 方法
Monkey m2=(Monkey) session.get(Monkey.class,new Long(1));
System.out.println(m1==m2); //true
tx.commit();
  
```

以下图 6-3 显示了第一次执行 Session 的 get() 方法的过程。get() 方法先到 Session 缓存中查找 OID 为 1 的 Monkey 对象，由于还不存在这样的 Monkey 对象，因此只好通过 SQL select 语句到数据库中去加载该对象，并把它放到 Session 缓存中。

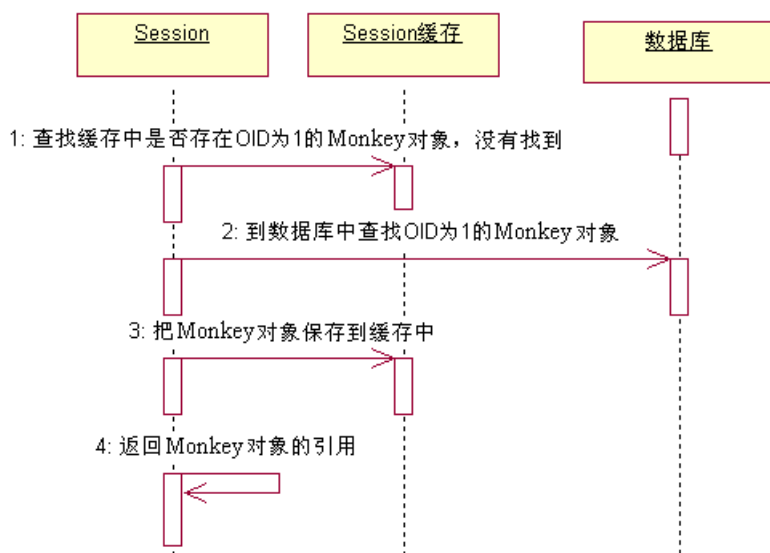


图 6-3 第一次执行 Session 的 get()方法的过程

以下图 6-4 显示了第二次执行 Session 的 get()方法的过程。get()方法先到 Session 缓存中查找 OID 为 1 的 Monkey 对象，由于已经存在这样的 Monkey 对象，就直接返回该 Monkey 对象的引用。因此在以上程序代码中，变量 m1 和变量 m2 引用的是同一个 Monkey 对象。

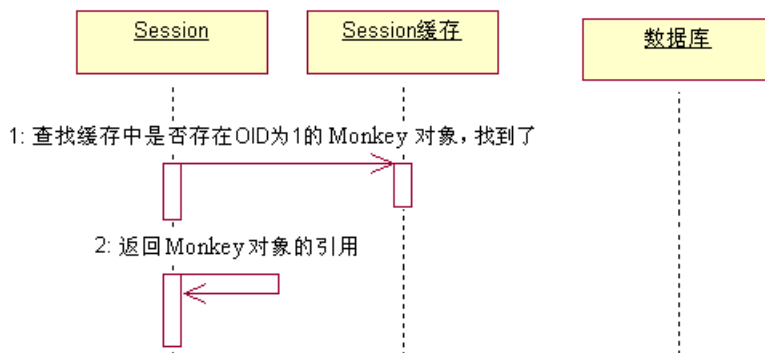


图 6-4 第二次执行 Session 的 get()方法的过程

(2) 当缓存中的持久化对象之间存在循环关联关系时，Session 会保证不出现访问对象图的死循环，以及由死循环引起的 JVM 堆栈溢出异常。

(3) 保证数据库中的相关记录与缓存中的相应对象保持同步。如图 6-5 所示，对象-关系映射文件建立了 MONKEYS 表与 Monkey 类的静态映射，而 Session 则建立了 MONKEYS 表中的关系数据与程序运行时的 Session 缓存中的 Monkey 对象的动态映射。

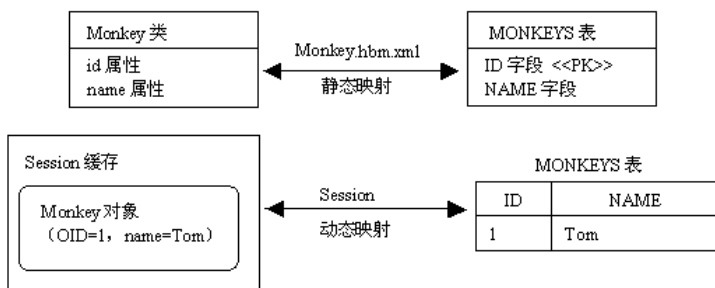


图 6-5 静态映射和动态映射

以下程序代码先把 Monkey 对象加载到 Session 缓存中，接下来修改 Session 缓存中 Monkey 对象的 name 属性：

```
Transaction tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
monkey.setName("Jack"); //修改 Monkey 对象的 name 属性
tx.commit();
```

如图 6-6 所示，执行完 monkey.setName("Jack") 语句后，Session 缓存中 Monkey 对象的 name 属性与 MONKEYS 表中对应记录的 NAME 字段的值不一致了。



图 6-6 Session 缓存中对象与数据库中相应记录不匹配

幸运的是，Session 在清理缓存的时候，会自动进行脏检查（dirty-check），如果发现 Session 缓存中的对象与数据库中相应记录不一致，就会根据对象的最新属性去同步更新数据库。在本例中，Session 会向数据库提交一条 update 语句，把 MONKEYS 表中对应记录的 NAME 字段的值改为“Jack”。

### 6.1.2 脏检查及清理缓存的机制

Session 到底是如何进行脏检查的呢？当一个 Monkey 对象被加入到 Session 缓存中时，Session 会为 Monkey 对象的值类型的属性复制一份快照（SnapShot）。当 Session 清理缓存时，会先进行脏检查，即比较 Monkey 对象的当前属性与它的快照，来判断 Monkey 对象的属性是否发生了变化，如果发生了变化，就称这个对象是“脏对象”，Session 会根据脏对象的最新属性来执行相关 SQL 语句，从而同步更新数据库。图 6-7 显示了 Session 加载 Monkey 对象、脏检查及同步更新数据库的过程。

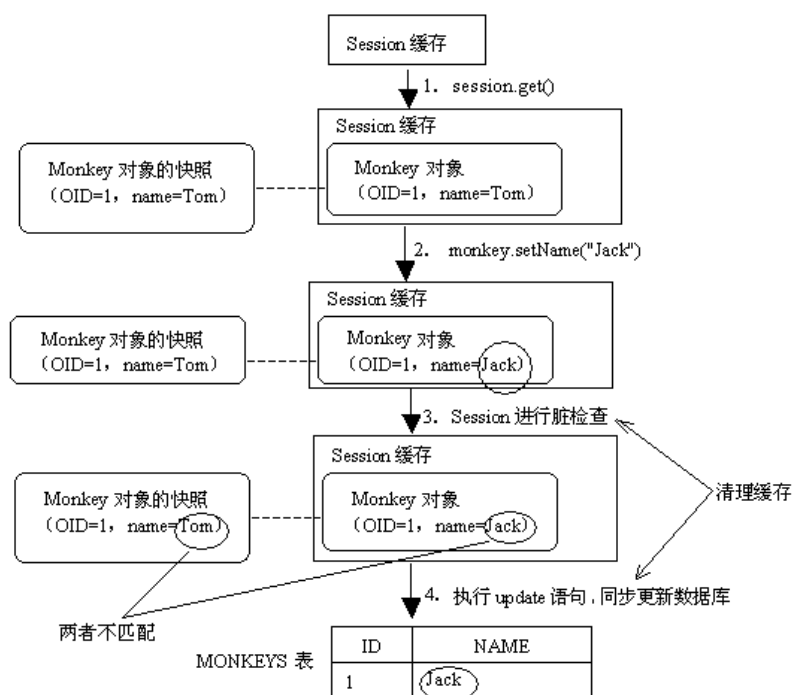


图 6-7 Session 加载 Monkey 对象、脏检查及同步更新数据库的过程

当 Session 缓存中对象的属性每次发生了变化, Session 并不会立即清理缓存及执行相关的 SQL update 语句, 而是在特定的时间点才清理缓存, 这使得 Session 能够把几条相关的 SQL 语句合并为一条 SQL 语句, 以便减少访问数据库的次数, 从而提高应用程序的数据访问性能。例如以下程序代码对 Monkey 对象的 name 属性修改了两次:

```
Transaction tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
monkey.setName("Jack");
monkey.setName("Mike");
tx.commit();
```

当 Session 清理缓存时, 不必先后执行两条 update 语句:

```
update MONKEYS set NAME= 'Jack'..... where ID=1;
update MONKEYS set NAME= 'Mike'..... where ID=1;
```

而只需执行一条 update 语句:

```
update MONKEYS set NAME= 'Mike'..... where ID=1;
```

Session 在清理缓存时, 按照以下顺序执行 SQL 语句。

- 按照应用程序调用 session.save()方法的先后顺序, 执行所有对实体进行插入的 insert 语句。

- 执行所有对实体进行更新的 `update` 语句。
- 执行所有对集合进行删除的 `delete` 语句。
- 执行所有对集合元素进行删除、更新或者插入的 `SQL` 语句。
- 执行所有对集合进行插入的 `insert` 语句。
- 按照应用程序调用 `session.delete()` 方法的先后顺序, 执行所有对实体进行删除的 `delete` 语句。

在默认情况下, `Session` 会在以下时间点清理缓存。

- 当应用程序调用 `org.hibernate.Transaction` 的 `commit()` 方法的时候, `commit()` 方法先清理缓存, 然后再向数据库提交事务。`Hibernate` 之所以把清理缓存的时间点安排在事务快结束时, 一方面是因为可以减少访问数据库的频率, 还有一方面是因为可以尽可能缩短当前事务对数据库中相关资源的锁定时间。
- 当应用程序执行一些查询操作时, 如果缓存中持久化对象的属性已经发生了变化, 就会先清理缓存, 使得 `Session` 缓存与数据库已进行了同步, 从而保证查询结果返回的是正确的数据。
- 当应用程序显式调用 `Session` 的 `flush()` 方法的时候。

例如以下代码两次清理了缓存。“`session.flush()`”语句使得 `Session` 立即清理缓存, 执行 `update` 语句, 把 `MONKEYS` 表的相应记录的 `NAME` 字段改为 “Jack”; “`tx.commit()`”语句会使得 `Session` 自动先清理缓存, 执行 `update` 语句, 把 `MONKEYS` 表的相应记录的 `NAME` 字段改为 “Mike”。

```
Transaction tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
monkey.setName("Jack");
session.flush(); //显式清理缓存, 执行 update 语句
monkey.setName("Mike");
tx.commit(); //自动清理缓存, 执行 update 语句, 再提交事务
```

### Tips

注意 `Session` 的 `commit()` 和 `flush()` 方法的区别。`flush()` 方法进行清理缓存的操作, 执行一系列的 `SQL` 语句, 但不会提交事务; `commit()` 方法会先调用 `flush()` 方法, 然后提交事务。提交事务意味着对数据库所做的更新被永久保存下来。

如果不希望 `Session` 在以上默认的时间点清理缓存, 也可以通过 `Session` 的 `setFlushMode()` 方法来显式设定清理缓存的时间点。`FlushMode` 类定义了 3 种不同的清理模式: `FlushMode.AUTO`、`FlushMode.COMMIT` 和 `FlushMode.NEVER`。例如, 以下代码用于把清理模式设为 `FlushMode.COMMIT`:

```
session.setFlushMode(FlushMode.COMMIT);
```

表 6-1 列出了 3 种清理模式各自执行清理缓存操作的时间点。

表 6-1 3 种清理模式

清理缓存的模式	各种查询方法	Transaction 的 commit()方法	Session 的 flush()方法
FlushMode.AUTO (默认模式)	清理	清理	清理
FlushMode.COMMIT	不清理	清理	清理
FlushMode.NEVER	不清理	不清理	清理

FlushMode.AUTO 是默认值，这也是优先考虑的清理模式，它会保证在整个事务中，Session 缓存中的对象和数据库数据保持一致。如果事务仅包含查询数据库的操作，而不会修改数据库的数据，也可以选用 FlushMode.COMMIT 模式，这可以避免在执行各种查询操作时先清理缓存，以稍微提高应用程序的性能。FlushMode.NEVER 模式意味着只有当程序显式调用 Session 的 flush()方法的时候才会清理缓存，这适用于需要长时间运行的复杂事务。

## 6.2 Java 对象在 Hibernate 持久化层的状态

当应用程序通过 new 语句创建了一个 Java 对象，这个对象的生命周期就开始了，当不再有任何引用变量引用它，这个对象就结束生命周期，它占用的内存就可以被 JVM 的垃圾回收器回收。

站在持久化层的角度，一个 Java 对象在它的生命周期中，可处于以下 4 个状态之一。

- 临时状态 (transient): 刚刚用 new 语句创建，还没有被持久化，并且不处于 Session 的缓存中。处于临时状态的 Java 对象被称为临时对象。
- 持久化状态 (persistent): 已经被持久化，并且加入到 Session 的缓存中。处于持久化状态的 Java 对象被称为持久化对象。
- 删除状态 (removed): 不再处于 Session 的缓存中，并且 Session 已经计划将其从数据库中删除。处于删除状态的 Java 对象被称为被删除对象。
- 游离状态 (detached): 已经被持久化，但不再处于 Session 的缓存中。处于游离状态的 Java 对象被称为游离对象。

### Tips

注意持久化类与持久化对象是不同的概念。持久化类的实例可以处于临时状态、持久化状态、删除状态和游离状态，其中处于持久化状态的实例被称为持久化对象。

在本书中，对象的状态有两种含义，一种含义是指由对象的属性表示的数据，一种含义是指以上的临时状态、持久化状态、删除状态或游离状态之一。读者应该根据上下文来辨别状态的具体含义，当文中提到 Session 按照对象的状态变化来同步更新数据库，这里的状态是指对象的属性表示的数据。



表 6-2 列出了 6.1 节的例程 6-1 中 Monkey 对象的状态转换过程。

程序代码	Monkey 对象的生命周期	Monkey 对象的状态
tx = session.beginTransaction();	开始生命周期	临时状态
Monkey m1=new Monkey("Tom",new HashSet());		
session.save(m1);	处于生命周期中	转变为持久化状态
Long id=m1.getId();	处于生命周期中	处于持久化状态
m1=null;		
Monkey m2=(Monkey)session.get(Monkey.class,id);		
tx.commit();		
session.close();	处于生命周期中	转变为游离状态
System.out.println(m2.getName());	处于生命周期中	处于游离状态
m2=null;	结束生命周期	结束生命周期

从表 6-2 看出, Session 的 save()方法使 Monkey 对象由临时状态转变为持久化状态, close()方法使 Monkey 对象由持久化状态转变为游离状态。图 6-8 为 Java 对象的完整状态转换图, Session 的特定方法使 Java 对象由一个状态转换到另一个状态。从图 6-8 看出, 当 Java 对象处于临时状态、删除状态或游离状态, 只要不被任何变量引用, 就会结束生命周期, 它占用的内存就可以被 JVM 的垃圾回收器回收; 当处于持久化状态, 由于 Session 的缓存会引用它, 因此它始终处于生命周期中。

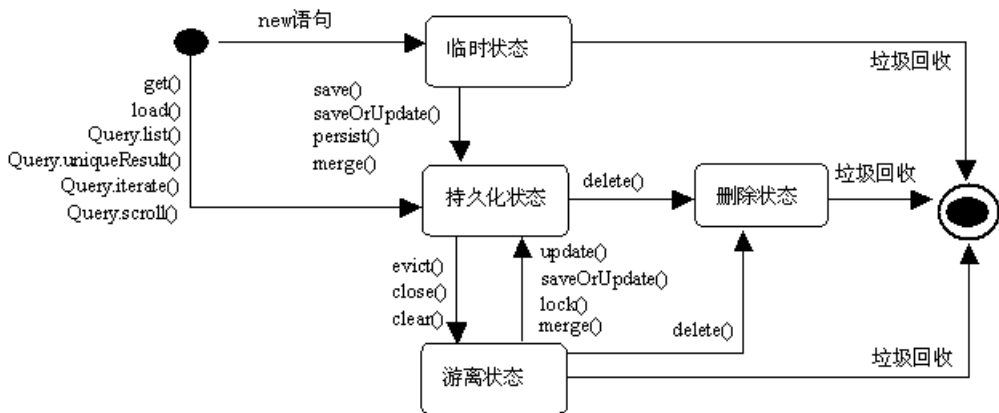


图 6-8 对象在 Hibernate 持久化层的状态转换图

### 6.2.1 临时对象的特征

临时对象具有以下特征。

- 在使用代理主键的情况下，OID 通常为 null。
- 不处于 Session 的缓存中，也可以说，不被任何一个 Session 实例关联。
- 在数据库中没有对应的记录。

当通过 new 语句刚创建了一个 Java 对象，它处于临时状态，此时不和数据库中的任何记录对应。

### 6.2.2 持久化对象的特征

持久化对象具有以下特征。

- OID 不为 null。
- 位于一个 Session 实例的缓存中，也可以说，持久化对象总是被一个 Session 实例关联。
- 持久化对象和数据库中的相关记录对应。
- Session 在清理缓存时，会根据持久化对象的属性变化，来同步更新数据库。

Session 的许多方法都能够使 Java 对象进入持久化状态：

- Session 的 save()方法把临时对象转变为持久化对象。
- Session 的 load()或 get()方法返回的对象总是处于持久化状态。
- Query 的 list()方法返回的 List 集合中存放的都是持久化对象。
- Session 的 update()、saveOrUpdate()和 lock()方法使游离对象转变为持久化对象。
- 当一个持久化对象关联一个临时对象，在允许级联保存的情况下，Session 在清理缓存时会把这个临时对象也转变为持久化对象。

Hibernate 保证在同一个 Session 实例的缓存中，数据库表中的每条记录只对应唯一的持久化对象。例如对于以下代码，共创建了两个 Session 实例：session1 和 session2。session1 和 session2 拥有各自的缓存。在 session1 的缓存中，只会有唯一的 OID 为 1 的 Monkey 持久化对象，在 session2 的缓存中，也只会有一唯一的 OID 为 1 的 Monkey 持久化对象。因此在内存中共有两个 Monkey 持久化对象，一个属于 session1 的缓存，一个属于 session2 的缓存。引用变量 a 和 b 都引用 session1 缓存中的 Monkey 持久化对象，而引用变量 c 引用 session2 缓存中的 Monkey 持久化对象：

```
Session session1=sessionFactory.openSession();
Session session2=sessionFactory.openSession();
Transaction tx1 = session1.beginTransaction();
Transaction tx2 = session2.beginTransaction();

Monkey a=(Monkey)session1.get(Monkey.class,new Long(1));
```

```

Monkey b=(Monkey)session1.get(Monkey.class,new Long(1));
Monkey c=(Monkey)session2.get (Monkey.class,new Long(1));

System.out.println(a==b); //true
System.out.println(a==c); //false

tx1.commit();
tx2.commit();
session1.close();
session2.close();

```

### 6.2.3 被删除对象的特征

被删除对象具有以下特征。

- OID 不为 null。
- 从一个 Session 实例的缓存中删除。
- 被删除对象和数据库中的相关记录对应。
- Session 已经计划将其从数据库中删除。
- Session 在清理缓存时, 会执行 SQL delete 语句, 删除数据库中的相应记录。
- 一般情况下, 应用程序不应该再使用被删除的对象。

在以下情况, Java 对象进入删除状态:

- Session 的 delete() 方法把持久化对象及游离对象转变为被删除对象。
- 当一个持久化对象 A 关联一个持久化对象 B, 在允许级联删除的情况下, Session 删除持久化对象 A 时, 会级联删除持久化对象 B, 使得持久化对象 A 和持久化对象 B 都进入删除状态。

### 6.2.4 游离对象的特征

游离对象具有以下特征。

- OID 不为 null。
- 不再位于 Session 的缓存中, 也可以说, 游离对象不被 Session 关联。
- 游离对象是由持久化对象转变过来的, 因此在数据库中可能还存在与它对应的记录 (前提条件是没有其他程序删除了这条记录)。

游离对象与临时对象的相同之处在于: 两者都不被 Session 关联, 因此 Hibernate 不会保证它们的属性变化与数据库保持同步。游离对象与临时对象的区别在于: 前者是由持久化对象转变过来的, 因此可能在数据库中还存在对应的记录, 而后者在

数据库中没有对应的记录。

游离对象与被删除对象的相同之处在于：两者都不位于 Session 的缓存中，并且在数据库中都可能存在对应的记录。两者的区别在于，游离对象与 Session 完全脱离关系，而对于被删除对象，Session 会计划将其从数据库中删除，等到 Session 清理缓存时，会执行相应的 SQL delete 语句，从数据库中删除对应的记录。

Session 的以下方法使持久化对象转变为游离对象。

- 当调用 Session 的 close()方法时，Session 的缓存被清空，缓存中的所有持久化对象都变为游离对象。如果在应用程序中没有引用变量引用这些游离对象，它们就会结束生命周期。
- Session 的 evict()方法能够从缓存中清除一个持久化对象，使它变为游离对象。当 Session 的缓存中保存了大量的持久化对象，会消耗许多内存空间，为了提高性能，可以考虑调用 evict()方法，从缓存中清除一些持久化对象。但是在多数情况下不推荐使用 evict()方法，而应该通过查询语言或者显式的导航来控制对象图的深度。
- Session 的 clear()方法能够清出缓存中的所有持久化对象，使它们变为游离对象。

## 6.3 Session 接口的详细用法

Session 接口是 Hibernate 向应用程序提供的操纵数据库的最主要的接口，它提供了基本的保存、更新、删除和加载等方法。

### 6.3.1 Session 的 save()方法

Session 的 save()方法使一个临时对象转变为持久化对象。例如以下代码保存一个 Monkey 对象：

```
Monkey monkey=new Monkey(); //创建 Monkey 临时对象
monkey.setName("Tom");
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(monkey); //计划执行 insert 语句
System.out.println(monkey.getId()); //id=1
tx.commit(); //真正执行 insert 语句
session.close();
```

Session 的 save()方法完成以下操作。

(1) 把 Monkey 对象加入到 Session 缓存中, 使它进入持久化状态。

(2) 选用映射文件指定的标识符生成器为持久化对象分配唯一的 OID。

Monkey.hbm.xml 文件中<id>元素的<generator>子元素指定标识符生成器:

```
<id name="id" column="ID">
    <generator class="increment"/>
</id>
```

(3) 计划执行一个 insert 语句, 把 Monkey 对象当前的属性值组装到 insert 语句中:

```
insert into MONKEYS(ID,NAME,.....) values(1, 'Tom',...);
```

值得注意的是, save()方法并不立即执行 SQL insert 语句。只有当 Session 清理缓存时, 才会执行 SQL insert 语句。如果在 save()方法之后, 又修改了持久化对象的属性, 这会使得 Session 在清理缓存时, 额外执行 SQL update 语句。以下两段代码尽管都能完成相同的功能, 但是左边代码仅执行一条 SQL insert 语句, 而右边代码执行一条 SQL insert 和一条 SQL update 语句。左边的代码减少了操纵数据库的次数, 具有更好的运行性能。

```
Monkey monkey=new Monkey();
```

```
//先设置 Monkey 对象的属性, 再保存它
```

```
monkey.setName("Tom");
```

```
session.save(monkey);
```

```
tx.commit(); //执行 insert 语句
```

```
Monkey monkey=new Monkey();
```

```
//先保存 Monkey 对象, 再修改它的属性
```

```
session.save(monkey);
```

```
monkey.setName("Tom");
```

```
tx.commit(); //执行 insert 和 update 语句
```

### Tips

在使用代理主键的场合, 无论 Java 对象处于临时状态、持久化状态、删除状态还是游离状态, 应用程序都不应该修改它的 OID。因此, 比较安全的做法是, 在定义持久化类时, 把它的 setId()方法设为 private 类型, 禁止外部程序访问该方法。

Session 的 save()方法是用来持久化一个临时对象的。在应用程序中不应该把持久化对象或游离对象传给 save()方法。例如以下代码两次调用了 Session 的 save()方法, 第二次传给 save()方法的 Monkey 对象处于持久化状态, 这步操作其实是多余的:

```
Monkey monkey=new Monkey();
session.save(monkey);
monkey.setName("Tom");
session.save(monkey); //这步操作是多余的
tx.commit();
```

再例如以下代码把 Monkey 游离对象传给 session2 的 save()方法, session2 会把它当做临时对象处理, 再次向数据库中插入一条 Monkey 记录:

```
Monkey monkey=new Monkey();
monkey.setName("Tom");
```

```

Session session1=sessionFactory.openSession();
Transaction tx1 = session1.beginTransaction();
session1.save(monkey) ; //此时 Monkey 对象的 ID 变为 1
tx1.commit();
session1.close(); //此时 Monkey 对象变为游离对象
Session session2=sessionFactory.openSession();
Transaction tx2 = session2.beginTransaction();
session2.save(monkey) ; //此时 Monkey 对象的 ID 变为 2
tx2.commit();
session2.close();

```

尽管以上程序代码能正常运行，但是会导致 MONKEYS 表中有两条代表相同业务实体的记录，因此不符合业务逻辑。所以让 Session 的 save()方法保存游离对象是不符合持久化规范的操作。

### 6.3.2 Session 的 load()和 get()方法

Session 的 load()和 get()方法都能根据给定的 OID 从数据库中加载一个持久化对象，这两个方法的一个区别在于：当数据库中不存在与 OID 对应的记录时，load()方法抛出 org.hibernate.ObjectNotFoundException 异常，而 get()方法返回 null。

load()和 get()方法的一个更重要的区别在于两者采用不同的检索策略。如果读者不了解检索策略的概念，建议先阅读本书第 7 章（Hibernate 的检索策略和检索方式），了解了检索策略的概念后再来阅读以下这段内容。在默认情况下，所有的持久化类都采用延迟检索策略，即映射文件中<class>元素的 lazy 属性取默认值“true”：

```
<class name="mypack.Monkey" table="MONKEYS" lazy="true" >
```

在默认情况下，load()方法会采用延迟检索策略加载持久化对象。除非把<class>元素的 lazy 属性的值设为“false”，load()方法才会采用立即检索策略：

```
<class name="mypack.Monkey" table="MONKEYS" lazy="false" >
```

而 get()方法会忽略<class>元素的 lazy 属性，即不管 lazy 属性取什么值，get()方法总是采用立即检索策略。

假定<class>元素的 lazy 属性取默认值“true”，则 load()方法采用延迟检索策略，而 get()方法采用立即检索策略，此时这两个方法有各自的使用场合：

- (1) 如果加载一个对象的目的是为了访问它的各个属性，可以用 get()方法。
- (2) 如果加载一个对象的目的是为了删除它，或者为了建立与别的对象的关联关系，可以用 load()方法。例如假定有一个 Monkey 对象和 Team 对象尚未建立关联关系，以下代码试图建立两者的多对一单向关联关系：

```

Transaction tx = session.beginTransaction();
//立即检索策略
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
//延迟检索策略
Team team=(Team)session.load(Team.class,new Long(1));
monkey.setTeam(team); //建立 Monkey 和 Team 的多对一单向关联关系
tx.commit(); //执行 update 语句

```

Session 不需要知道 Team 对象的各个属性的值,而只要知道 Team 对象的 OID,就可以生成以下 update 语句:

```
update MONKEYS set TEAM_ID=1,NAME= ... where ID=1;
```

由 get()、load()或其他查询方法返回的对象都位于当前 Session 的缓存中,处于持久化状态,因此接下来修改了持久化对象的属性后,当 Session 清理缓存时,会根据持久化对象的属性变化来同步更新数据库。

### 6.3.3 Session 的 update()方法

Session 的 update()方法使一个游离对象转变为持久化对象,并且会计划执行一条 update 语句。例如以下代码在 session1 中保存一个 Monkey 对象,然后在 session2 中更新这个 Monkey 对象:

```

Monkey monkey=new Monkey();
monkey.setName("Tom");
Session session1=sessionFactory.openSession();
Transaction tx1 = session1.beginTransaction();
session1.save(monkey);
tx1.commit(); //清理缓存,提交事务
session1.close(); //此时 Monkey 对象变为游离对象

Session session2=sessionFactory.openSession();
Transaction tx2 = session2.beginTransaction();
monkey.setName("Linda") //在和 session2 关联之前修改 Monkey 对象的属性
session2.update(monkey); //Monkey 对象和 session2 关联
monkey.setName("Jack"); //在和 session2 关联之后修改 Monkey 对象的属性
tx2.commit(); //清理缓存,提交事务
session2.close();

```

Session 的 update()方法完成以下操作:

- (1) 把 Monkey 游离对象加入到当前 Session 缓存中,使它变为持久化对象。
- (2) 计划执行一个 update 语句。值得注意的是,Session 只有在清理缓存的时候才会执行 update 语句,并且在执行时才会把 Monkey 对象当前的属性值组装到

update 语句中。因此，即使程序中多次修改了 Monkey 对象的属性，在清理缓存时只会执行一次 update 语句。以下两段代码是等价的，无论是左边的代码，还是右边的代码，Session 都只会执行一条 update 语句：

<pre>..... Session session2=sessionFactory.openSession(); Transaction tx2 = session2.beginTransaction(); <b>monkey.setName("Linda")</b> <b>session2.update(monkey);</b> <b>monkey.setName("Jack");</b> tx2.commit(); //清理缓存，提交事务 session2.close();</pre>	<pre>..... Session session2=sessionFactory.openSession(); Transaction tx2 = session2.beginTransaction(); <b>session2.update(monkey);</b> <b>monkey.setName("Linda")</b> <b>monkey.setName("Jack");</b> tx2.commit(); //清理缓存，提交事务 session2.close();</pre>
--	--

以上代码尽管把 Monkey 对象的 name 属性修改了两次，但 Session 在清理缓存时，根据 Monkey 对象的当前属性来组装 update 语句，因此执行的 update 语句为：

```
update MONKEYS set name='Jack' ..... where ID=1;
```

只要通过 update() 方法使游离对象被一个 Session 关联，即使没有修改 Monkey 对象的任何属性，Session 在清理缓存时也会执行由 update() 方法计划的 update 语句。例如以下程序使 Monkey 对象被 session2 关联，但是没有修改 Monkey 对象的任何属性：

```
//假定 monkey 对象为游离对象
.....
Session session2=sessionFactory.openSession();
Transaction tx2 = session2.beginTransaction();
session2.update(monkey);
tx2.commit();
session2.close();
```



Session 在清理缓存时，会执行由 update()方法计划的 update 语句，并且根据 Monkey 对象的当前属性来组装 update 语句：

```
update MONKEYS set name='Tom' ..... where ID=1;
```

如果希望 Session 仅当修改了 Monkey 对象的属性时，才执行 update 语句，可以把映射文件中<class>元素的 select-before-update 设为 true，该属性的默认值为 false：

```
<class name="mypack.Monkey" table="MONKEYS"  
      select-before-update="true">
```

如果按以上方式修改了 Monkey.hbm.xml 文件，当 Session 清理缓存时，会先执行一条 select 语句：

```
select * from MONKEYS where ID=1;
```

然后比较 Monkey 对象的属性是否和从数据库中检索出来的记录一致，只有在不一致的情况下，才执行 update 语句。

应该根据实际情况来决定是否应该把 select-before-update 设为 true。如果 Java 对象的属性不会经常变化，可以把 select-before-update 属性设为 true，避免 Session 执行不必要的 update 语句，这样会提高应用程序的性能。如果程序需要经常修改 Java 对象的属性，就没必要把这个 select-before-update 属性设为 true，因为它会导致在执行 update 语句之前，执行一条多余的 select 语句。

在分层的软件结构中，临时对象和游离对象会在客户层与业务逻辑层之间传递。例如对于花果山猴子管理信息系统会包含以下业务过程。

(1) 创建猴子对象：客户层创建一个 Monkey 临时对象，里面包含用户输入的猴子信息，业务逻辑层调用 Session 的 save()方法持久化这个临时对象。

(2) 查询猴子对象：业务逻辑层按照客户层给定的查询条件，调用 Query 接口的 list()方法，查询出符合条件的 Monkey 对象，向客户层返回处于游离状态的 Monkey 对象。

(3) 修改猴子对象：客户层修改步骤(2)返回的 Monkey 游离对象的属性，然后再把它传给业务逻辑层，业务逻辑层调用 Session 的 update()方法更新数据库。

(4) 删除猴子对象：客户层把步骤(2)返回的 Monkey 游离对象再传给业务逻辑层，业务逻辑层调用 Session 的 delete()方法从数据库中删除相关的记录。

在业务逻辑层，每当事务结束，都会关闭 Session，使 Monkey 对象进入游离状态。图 6-9 显示了客户层与业务逻辑层之间传递临时对象和游离对象的过程。

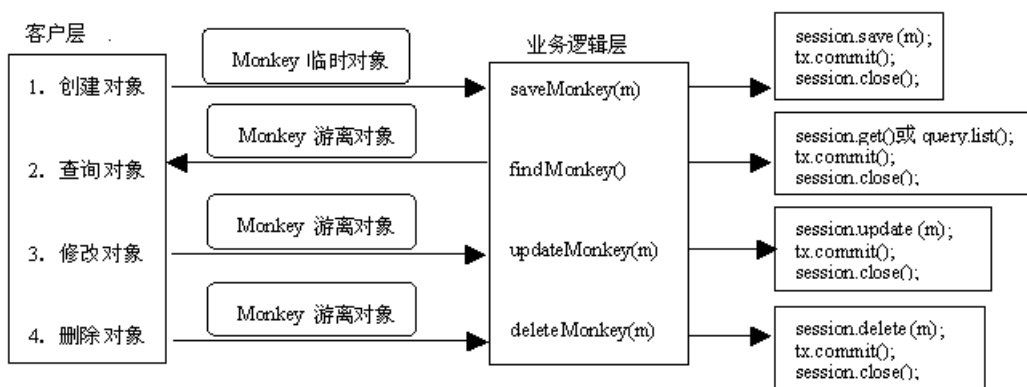


图 6-9 客户层与业务逻辑层之间传递临时对象和游离对象的过程

### 6.3.4 Session 的 saveOrUpdate()方法

Session 的 saveOrUpdate()方法同时包含了 save()与 update()方法的功能，如果传入的参数是临时对象，就调用 save()方法；如果传入的参数是游离对象，就调用 update()方法；如果传入的参数是持久化对象，那就直接返回。在传入参数不是持久化对象的前提下，saveOrUpdate()方法如何判断一个对象处于临时状态还是游离状态呢？如果满足以下情况之一，Hibernate 就把它作为临时对象，否则就作为游离对象。

- Java 对象的 OID 取值为 null。
- Java 对象具有 version 版本控制属性并且取值为 null。
- 在映射文件中为<id>元素设置了 unsaved-value 属性，并且 Java 对象的 OID 取值与这个 unsaved-value 属性值匹配。
- 在映射文件中为 version 版本控制属性设置了 unsaved-value 属性，并且 Java 对象的 version 版本控制属性的取值与映射文件中 unsaved-value 属性值匹配。

在以下程序中，假定 monkeyA 起初为游离对象，monkeyB 起初为临时对象，session2 的 saveOrUpdate()方法分别将它们变为持久化对象：

```
//假定 monkeyA 为游离对象
.....
Session session2=sessionFactory.openSession();
Transaction tx2 = session2.beginTransaction();
Monkey monkeyB=new Monkey(); //创建临时对象
monkeyB.setName("Tom");

session2.saveOrUpdate(monkeyA); //使 monkeyA 游离对象被 session2 关联
session2.saveOrUpdate(monkeyB); //使 monkeyB 临时对象被 session2 关联
```

```
tx2.commit();
session2.close();
```

如果 Monkey 类的 id 属性为 java.lang.Long 类型，它的默认值为 null，那么 session2 很容易就判断出 monkeyA 对象为游离对象，因为它的 id 属性不为 null；而 monkeyB 对象为临时对象，因为它的 id 属性为 null。因此 session2 对 monkeyA 对象执行 update 操作，对 monkeyB 对象执行 insert 操作。

如果 Monkey 类的 id 属性为 long 类型，它的默认值为 0，此时需要显式设置 <id> 元素的 unsaved-value 属性，它的默认值为 null：

```
<id name="id" column="ID" unsaved-value="0">
    <generator class="increment"/>
</id>
```

这样，如果一个 Monkey 对象的 id 取值为 0，saveOrUpdate() 方法就会把它作为临时对象。

### 6.3.5 Session 的 merge() 方法

Session 的 merge() 方法能够把一个游离对象的属性复制到一个持久化对象中。当 Session 用 update() 方法关联一个游离对象时，如果在 Session 缓存中已经存在一个同类型的并且 OID 相同的持久化对象，那么 update() 方法会抛出 NonUniqueException：

```
monkey1.setName("Jack");//假定 monkey1 为游离对象，OID 为 1

Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
//session 加载 OID 为 1 的 Monkey 持久化对象
Monkey monkey2=(Monkey) session.get(Monkey.class,new Long(1));
//把 OID 为 1 的 monkey1 游离对象加入到 session 的缓存中
session.update(monkey1); //抛出 NonUniqueException

tx.commit();
session.close();
```

下面的代码把 update() 方法改为 merge() 方法：

```
monkey1.setName("Jack");//假定 monkey1 为游离对象，OID 为 1

Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
//session 加载 OID 为 1 的 Monkey 持久化对象
Monkey monkey2=(Monkey) session.get(Monkey.class,new Long(1));
//把 monkey1 对象的属性复制到 Session 缓存中的相应持久化对象中
```

```
Monkey monkey3=(Monkey) session.merge(monkey1);

monkey1==monkey2; //false
monkey1==monkey3; //false
monkey2==monkey3; //true

tx.commit(); //执行 update 语句,把 MONKEYS 表中 ID 为 1 记录的 NAME 字段改为 Jack
session.close();
return monkey3;
```

如图 8-13 所示, Session 的 merge()方法的处理流程如下:

(1)根据 monkey1 游离对象的 OID 到 Session 缓存中查找匹配的持久化对象。在本例中,找到了匹配的 monkey2 持久化对象,就把 monkey1 游离对象的属性复制到 monkey2 持久化对象中,计划执行一条 update 语句,再返回 monkey2 持久化对象的引用,所以表达式“monkey2==monkey3”的值为 true。

(2)如果在 Session 缓存中没有找到与 monkey1 游离对象的 OID 一致的 Monkey 持久化对象,那么就试图根据这个 OID 从数据库中加载 Monkey 持久化对象。如果在数据库中存在这样的 Monkey 持久化对象,就把 monkey1 游离对象的属性复制到这个刚加载的 Monkey 持久化对象中,计划执行一条 update 语句,再返回这个 Monkey 持久化对象的引用。如果在数据库中不存在这样的 Monkey 持久化对象,就会创建一个新的 Monkey 对象,把 monkey1 游离对象的属性复制到这个新建的 Monkey 对象中,再调用 save()方法持久化这个 Monkey 对象,最后返回这个 Monkey 持久化对象的引用。

(3)如果 merge()方法的参数 monkey1 为一个临时对象,那么也会创建一个新的 Monkey 对象,把 monkey1 临时对象的属性复制到这个新建的 Monkey 对象中,再调用 save()方法持久化这个 Monkey 对象,最后返回这个 Monkey 持久化对象的引用。

从 merge()方法的处理流程可以看出, merge()方法返回的 monkey3 是一个持久化对象。参数传入的 monkey1 为游离对象或临时对象, monkey1 的属性被复制到 monkey3 持久化对象中。程序调用完 merge()方法, monkey1 对象就没有使用价值,可以结束生命周期了,程序接下来可以继续操纵 monkey3 对象。

### 6.3.6 Session 的 delete()方法

Session 的 delete()方法用于从数据库中删除一个 Java 对象。delete()方法既可以删除持久化对象,也可以删除游离对象。delete()方法的处理过程如下。

(1) 如果传入的参数是游离对象, 先使游离对象被当前 Session 关联, 使它变为持久化对象。如果传入的参数是持久化对象, 则忽略这一步。

(2) 计划执行一个 delete 语句。

(3) 把对象从 Session 缓存中删除, 该对象进入删除状态。

值得注意的是, Session 只有在清理缓存的时候才会执行 delete 语句。例如以下代码先加载一个持久化对象, 然后通过 delete() 方法将它删除:

```
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
//先加载一个持久化对象,
//此处加载持久化对象是为了删除它, 无须访问它的属性,
//因此用 load() 方法而不是 get() 方法, 可以获得更好的性能
Monkey monkey=(Monkey) session.load(Monkey.class,new Long(1));
//计划执行一个 delete 语句, 从缓存中删除 Monkey 对象
//把 Monkey 对象转变为被删除对象
session.delete(monkey);

//以下 session.contains() 方法判断 Session 缓存中是否存在 monkey 对象
System.out.println(session.contains(monkey)); //打印 false
tx.commit();//执行 delete 语句
session1.close();
```

以下代码直接通过 delete() 方法删除一个游离对象:

```
Session session=sessionFactory.openSession();
Transaction tx = session.beginTransaction();
//假定 monkey 是一个游离对象
//以下 delete() 方法先使它被 Session 关联, 使它变为持久化对象,
//然后计划执行一个 delete 语句, 并使它变为被删除对象
session.delete(monkey);
tx.commit(); //执行 delete 语句
session.close(); //从缓存中删除 Monkey 对象
```

在默认情况下, delete() 方法能把持久化对象或游离对象转变为被删除对象, 被删除对象是无用对象, 程序不应该再使用这些对象。

Session 的 delete() 方法一次只能删除一个对象。本章 6.5.3 节 (通过 HQL 来进行批量操作) 将介绍批量删除多个 Monkey 对象的方法。

## 6.4 级联操纵对象图

对于实际应用, 对象与对象之间是相互关联的。因此, 在 Session 缓存中存放的是一幅相互关联的对象图。假定 Team 类与 Monkey 类双向关联。以下代码看似

仅加载了一个 Team 对象：

```
Team team=(Team)session.get(Team.class, new Long(1));
```

实际上，如果在关联级别设置了立即检索策略，那么 Session 可以自动加载所有和 Team 对象关联的 Monkey 对象，参见图 6-10。本书第 7 章将介绍立即检索策略的概念。

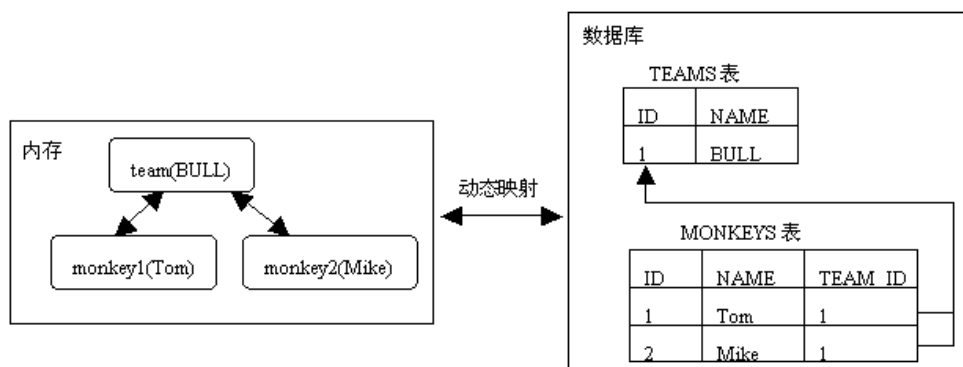


图 6-10 内存中互相关联的对象图与数据库中的关系数据对应

在应用程序中，可以通过 Monkey 对象的 getTeam() 方法导航到关联的 Team 对象，通过 Team 对象的 getMonkeys() 方法导航到关联的所有 Monkey 对象：

```
Team team=monkey1.getTeam();
Set monkeys=team.getMonkeys();
```

在对象-关系映射文件中，用于映射持久化类之间关联关系的元素，如 <set>、<many-to-one> 和 <one-to-one> 元素，都有一个 cascade 属性，它用于指定如何操纵与当前对象关联的其他对象。第 5 章已经介绍了 cascade 属性的一些可选值（如 save-update、delete 和 all-delete-orphan）的用法。表 6-3 列出了 cascade 属性的所有可选值。

表 6-3 cascade 属性

cascade 属性值	描 述
none	当 Session 操纵当前对象时，忽略其他关联的对象。它是 cascade 属性的默认值
save-update	当通过 Session 的 save()、update() 及 saveOrUpdate() 方法来保存或更新当前对象时，级联保存所有关联的新建的临时对象，并且级联更新所有关联的游离对象
delete	当通过 Session 的 delete() 方法删除当前对象时，会级联删除所有关联的对象
lock	当通过 Session 的 lock() 方法把当前游离对象加入到 Session 缓存中时，会把所有关联的游离对象也加入到 Session 缓存中。本书第 15 章（处理并发问题）在介绍利用 Hibernate 的版本控制来实现乐观锁时，将介绍 lock() 方法的用法
evict	当通过 Session 的 evict() 方法从 Session 缓存中清除当前对象时，会级联清除所有关联的对象
refresh	当通过 Session 的 refresh() 方法刷新当前对象时，会级联刷新所有关联的对象。所谓刷新是指读取数

	数据库中相应数据，然后根据数据库中的最新数据去同步更新 Session 缓存中的相应对象
all	包含 save-update、delete、lock、evict 及 refresh 的行为
delete-orphan	删除所有和当前对象解除关联关系的对象
all-delete-orphan	包含 all 和 delete-orphan 的行为

## 6.5 批量处理数据

为了避免 Session 的缓存长时间占用大量内存空间，通常，在一个 Session 对象的缓存中只存放数量有限的持久化对象，等到 Session 对象处理事务完毕，还要关闭 Session 对象，从而及时释放 Session 的缓存占用的内存。

批量处理数据是指在一个事务中处理大量数据。以下程序在一个事务中批量更新 MONKEYS 表中年龄大于零的所有记录的 AGE 字段：

```
Transaction tx = session.beginTransaction();
Iterator monkeys=
    session.createQuery("from Monkey m where m.age>0").list().iterator();
while(monkeys.hasNext()){
    Monkey monkey=(Monkey)monkeys.next();
    monkey.setAge(monkey.getAge()+1);
}

tx.commit();
session.close();
```

如果 MONKEYS 表中有 1 万条年龄大于零的记录，那么 Hibernate 会一下子加载 1 万个 Monkey 对象到内存。当执行 tx.commit()方法时，会清理缓存，Hibernate 执行 1 万条更新 MONKEYS 表的 update 语句：

```
update MONKEYS set AGE=? ... where ID=i;
update MONKEYS set AGE=? ... where ID=j;
.....
update MONKEYS set AGE=? ... where ID=k;
```

以上批量更新方式有两个缺点：

- 占用大量内存，必须把 1 万个 Monkey 对象先加载到内存，然后一一更新它们。
- 执行的 update 语句的数目太多，每个 update 语句只能更新一个 Monkey 对象，必须通过 1 万条 update 语句才能更新 1 万个 Monkey 对象，频繁地访问数据库，会大大降低应用的性能。

一般说来，应该尽可能避免在应用程序中进行批量操作，而应该在数据库中直

接进行批量操作，例如直接在数据库中执行用于批量更新或删除的 SQL 语句，如果批量操作的逻辑比较复杂，则可以通过直接在数据库中运行的存储过程来完成批量操作。

当然，在应用程序中也可以进行批量操作，主要有以下方式：

- (1) 通过 Session 来进行批量操作。
- (2) 通过 StatelessSession 来进行批量操作。
- (3) 通过 HQL 来进行批量操作。

### 6.5.1 通过 Session 来进行批量操作

Session 的 save() 及 update() 方法都会把处理的对象存放在自己的缓存中。如果通过一个 Session 对象来处理大量持久化对象，应该及时从缓存中清空已经处理完毕并且不会再访问的对象。具体的做法是在处理完一个对象或小批量对象后，立刻调用 flush() 方法清理缓存，然后再调用 clear() 方法清空缓存。

通过 Session 来进行批量操作会受到以下约束：

(1) 需要在 Hibernate 的配置文件中设置 JDBC 单次批量处理的数目，合理的取值通常为 10 到 50 之间，例如：

```
hibernate.jdbc.batch_size=20
```

在按照本节介绍的方法进行批量操作时，应该保证每次向数据库发送的批量 SQL 语句数目与这个 batch\_size 属性一致。

(2) 如果对象采用 “identity” 标识符生成器，则 Hibernate 无法在 JDBC 层进行批量插入操作。

(3) 进行批量操作时，建议关闭 Hibernate 的第二级缓存。因为假如打开了第二级缓存，那么所有在第一级缓存中所做的操作还要在第二级缓存中重复进行，这会影响程序运行性能。

Session 的缓存是 Hibernate 的第一级缓存，通常它是事务范围内的缓存，也就是说，每个事务都有单独的第一级缓存。SessionFactory 的外置缓存为 Hibernate 的第二级缓存，它是应用范围内的缓存，也就是说，所有事务都共享同一个第二级缓存。在任何情况下，Hibernate 的第一级缓存总是可用的。而在默认情况下，Hibernate 的第二级缓存是关闭的，此外也可以在 Hibernate 的配置文件中通过如下方式显式关闭第二级缓存：

```
hibernate.cache.use_second_level_cache=false
```



## 1. 批量插入数据

以下代码一共向数据库中插入十万条 MONKEYS 记录，单次批量插入 20 条 MONKEYS 记录：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Monkey monkey = new Monkey(.....);
    session.save(monkey);
    if ( i % 20 == 0 ) { //单次批量操作的数目为 20
        session.flush(); //清理缓存，执行批量插入 20 条记录的 SQL insert 语句
        session.clear(); //清空缓存中的 Monkey 对象
    }
}

tx.commit();
session.close();
```

在以上程序中，每次执行 `session.flush()` 方法，就会向数据库中批量插入 20 条记录。接下来 `session.clear()` 方法把 20 个刚保存的 Monkey 对象从缓存中清空。

## 2. 批量更新数据

进行批量更新时，如果一下子把所有对象加载到 Session 的缓存中，然后再在缓存中一一更新它们，显然是不可取的。为了解决这一问题，可以使用可滚动的结果集 `org.hibernate.ScrollableResults`，Query 的 `scroll()` 方法返回一个 `ScrollableResults` 对象。以下代码演示批量更新 Monkey 对象，该代码一开始利用 `ScrollableResults` 对象来加载所有的 Monkey 对象：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults monkeys= session.createQuery("from Monkey")
.scroll(ScrollMode.FORWARD_ONLY);

int count=0;
while ( monkeys.next() ) {
    Monkey monkey = (Monkey) monkeys.get(0);
    monkey.setAge(monkey.getAge()+1); //更新 Monkey 对象的 age 属性
    if ( ++count % 20 == 0 ) { //单次批量操作的数目为 20

        session.flush(); //清理缓存，执行批量更新 20 条记录的 SQL update 语句
        session.clear(); //清空缓存中的 Monkey 对象
    }
}
```

```
tx.commit();
session.close();
```

在以上代码中，Query 的 scroll()方法返回的 ScrollableResults 对象中实际上并不包含任何 Monkey 对象，它仅包含了用于在线定位数据库中 MONKEYS 记录的游标。只有当程序遍历访问 ScrollableResults 对象中的特定元素时，它才会到数据库中加载相应的 Monkey 对象。

### 6.5.2 通过 StatelessSession 来进行批量操作

Session 具有一个用于保持内存中对象与数据库中相应数据保持同步的缓存，位于 Session 缓存中的对象为持久化对象。但在进行批量操作时，把大量对象存放在 Session 缓存中会消耗大量内存空间。作为一种替代方案，可以采用无状态的 StatelessSession 来进行批量操作。

以下代码利用 StatelessSession 来进行批量更新操作：

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults monkeys = session.getNamedQuery("GetMonkeys")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( monkeys.next() ) {
    Monkey monkey = (Monkey) monkeys.get(0);
    monkey.setAge(monkey.getAge()+1); //在内存中更新 Monkey 对象的 age 属性;
    session.update(monkey); //立即执行 update 语句，更新数据库中相应 MONKEYS 记录
}

tx.commit();
session.close();
```

从形式上看，StatelessSession 与 Session 的用法有点相似。StatelessSession 与 Session 相比，有以下区别：

(1) StatelessSession 没有缓存，通过 StatelessSession 来加载、保存或更新后的对象都处于游离状态。

(2) StatelessSession 不会与 Hibernate 的第二级缓存交互。

(3) 当调用 StatelessSession 的 save()、update()或 delete()方法时，这些方法会立即执行相应的 SQL 语句，而不会仅计划执行一条 SQL 语句。

(4) StatelessSession 不会对所加载的对象自动进行脏检查。所以在以上程序中，修改了内存中 Monkey 对象的属性后，还需要通过 StatelessSession 的 update()

方法来更新数据库中的相应数据。

(5) `StatelessSession` 不会对关联的对象进行任何级联操作。举例来说, 通过 `StatelessSession` 来保存一个 `Monkey` 对象时, 不会级联保存与之关联的 `Team` 对象。

(6) 通过同一个 `StatelessSession` 对象两次加载 OID 为 1 的 `Monkey` 对象时, 会得到两个具有不同内存地址的 `Monkey` 对象, 例如:

```
StatelessSession session = sessionFactory.openStatelessSession();
Monkey m1=(Monkey) session.get(Monkey.class,new Long(1));
Monkey m2=(Monkey) session.get(Monkey.class,new Long(1));
System.out.println(m1==m2); //打印 false
```

### 6.5.3 通过 HQL 来进行批量操作

Hibernate 3 中的 HQL (Hibernate Query Language, Hibernate 查询语言) 不仅可以检索数据, 还可以用于进行批量更新、删除和插入数据。批量操作实际上直接在数据库中完成, 所处理的数据不会被保存在 `Session` 的缓存中, 因此不会占用内存空间。

`Query.executeUpdate()` 方法和 JDBC API 中的 `PreparedStatement.executeUpdate()` 很相似, 前者执行用于更新、删除和插入的 HQL 语句, 而后者执行用于更新、删除和插入的 SQL 语句。

#### 1. 批量更新数据

以下程序代码演示通过 HQL 来批量更新 `Monkey` 对象:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate =
    "update Monkey m set m.name = :newName where m.name = :oldName";
int updatedEntities = session.createQuery( hqlUpdate )
    .setString( "newName", "Mike" )
    .setString( "oldName", "Tom" )
    .executeUpdate();

tx.commit();
session.close();
```

以上程序代码向数据库发送的 SQL 语句为:

```
update MONKEYS set NAME="Mike" where NAME="Tom"
```

## 2. 批量删除数据

Session 的 delete()方法一次只能删除一个对象，不适合进行批量删除操作。以下程序代码演示通过 HQL 来批量删除 Monkey 对象：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Monkey m where m.name = :oldName";
int deletedEntities = session.createQuery( hqlDelete )
    .setString( "oldName", "Tom" )
    .executeUpdate();
tx.commit();
session.close();
```

以上程序代码向数据库提交的 SQL 语句为：

```
delete from MONKEYS where NAME="Tom"
```

## 3. 批量插入数据

插入数据的 HQL 语法为：

```
insert into EntityName properties_list select_statement
```

以上 EntityName 表示持久化类的名字，properties\_list 表示持久化类的属性列表，select\_statement 表示子查询语句。

HQL 只支持 insert into ... select ... 形式的插入语句，而不支持 “insert into ... values ... ” 形式的插入语句。

下面举例说明如何通过 HQL 来批量插入数据。假定有 MonkeyCopy 和 Monkey 类，它们都有 id 和 name 属性，与这两个类对应的表分别为 MONKEY\_COPYYS 和 MONKEYS 表。MonkeyCopy.hbm.xml 和 Monkey.hbm.xml 文件分别为这两个类的映射文件。以下代码能够把 MONKEYS 表中的数据复制到 MONKEY\_COPYYS 表中：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into MonkeyCopy(id, name)"
    + " select m.id, m.name from Monkey m where m.id>1";
int createdEntities = s.createQuery( hqlInsert )
    .executeUpdate();
tx.commit();
session.close();
```

以上程序代码向数据库提交的 SQL 语句为：

```
insert into MONKEY_COPYYS(ID,NAME) select ID,NAME from MONKEYS where ID>1
```

## 6.6 Hibernate 的二级缓存结构

缓存是计算机领域非常通用的概念，它介于应用程序和永久性数据存储源（如硬盘上的文件或者数据库）之间，其作用是降低应用程序直接读写永久性数据存储源的频率，从而提高应用的运行性能。缓存中的数据是数据存储源中数据的备份，应用程序在运行时直接读写缓存中的数据，只在某些特定时刻按照缓存中的数据来同步更新数据存储源。图 6-11 显示了缓存在软件系统中的位置。

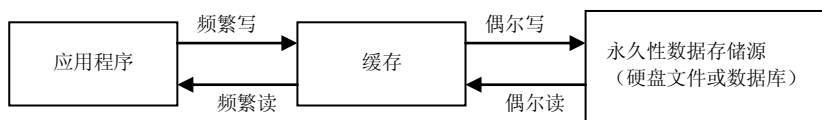


图 6-11 缓存在软件系统中的位置

缓存的物理介质通常是内存，而永久性数据存储源的物理介质通常是硬盘或磁盘，应用程序读写内存的速度显然比读写硬盘的速度快。如果缓存中存放的数据量非常大，也会用硬盘作为缓存的物理介质。

Hibernate 提供了两级缓存，如图 6-12 所示，第一级缓存是 Session 的缓存。由于 Session 对象的生命周期通常对应一个数据库事务或者一个应用事务，因此它的缓存是事务范围的缓存。第一级缓存是必需的，不允许而且事实上也无法被卸除。在第一级缓存中，持久化类的每个实例都具有唯一的 OID。

第二级缓存是一个可插拔的缓存插件，它由 SessionFactory 负责管理。由于 SessionFactory 对象的生命周期和应用程序的整个进程对应，因此第二级缓存是进程范围或集群范围的缓存。这个缓存中存放的是对象的散装数据。

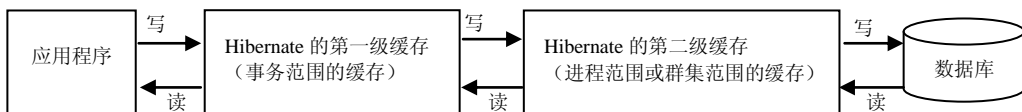


图 6-12 Hibernate 的二级缓存机制

在默认情况下，Hibernate 并没有启用 Hibernate 的第二级缓存。如果需要启用第二级缓存，需要进行专门的配置。如果读者想了解如何配置第二级缓存，可以参考本作者的另一本书《精通 Hibernate：Java 对象持久化技术详解》。

## 6.7 小结

站在持久化层的角度，Java 对象在生命周期中可处于临时状态、持久化状态、

删除状态和游离状态。处于持久化状态的 Java 对象位于一个 Session 实例的缓存中，Session 能根据这个对象的属性变化来同步更新数据库。Session 的 save()方法把一个临时对象转变为持久化对象，update()方法把一个游离对象转变为持久化对象，saveOrUpdate()方法先判断对象的状态，如果处于临时状态，就调用 save()方法，如果处于游离状态，就调用 update()方法。只有了解对象所处的状态，才能正确地通过 Hibernate API 来操纵它们。

更新数据库中的对象有两种方式：

(1) 先加载持久化对象，修改持久化对象的属性，然后 Session 在清理缓存时自动同步更新数据库中的相应数据：

```
Transaction tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
monkey.setName("Jack"); //修改 Monkey 持久化对象的 name 属性
tx.commit(); //清理 Session 缓存，更新数据库中的相应数据
```

(2) 更新游离对象，然后通过 Session 的 update()方法更新数据库中的相应数据：

```
Monkey monkey=... //假定 Monkey 为游离对象
monkey.setName("Jack"); //修改 Monkey 游离对象的 name 属性
Transaction tx = session.beginTransaction();
session.update(monkey); //计划执行一条 SQL update 语句
tx.commit(); //清理 Session 缓存，更新数据库中的相应数据
```

在映射文件中，cascade 属性用来指定如何操纵与当前对象关联的其他对象。例如如果把 cascade 属性设为 save-update，那么当通过 Session 的 save()、update() 及 saveOrUpdate()方法来保存或更新当前对象时，会级联保存所有关联的新建的临时对象，并且级联更新所有关联的游离对象。

## 第7章 Hibernate的检索策略和检索方式

现在，悟空已经能够熟练地把猴子及关联的武术队信息保存到数据库中，在需要的时候，还能把它们再加载到 Session 的缓存中。猴子及关联的武术队在 Session 的缓存中表现为相互关联的对象图。

可是，悟空在检索数据时遇到了一个会影响程序性能的问题：当 Hibernate 从数据库中加载 Team 对象时，如果同时自动加载所有关联的 Monkey 对象，而程序实际上仅需要访问 Team 对象，那么这些关联的 Monkey 对象就白白浪费了许多内存空间。

本章先以 Team 和 Monkey 类为例，介绍如何设置 Hibernate 的检索策略，以优化检索性能，解决悟空所面临的上述问题。

假定 MONKEYS 表的 TEAM\_ID 外键允许为 null，图 7-1 列出了 TEAMS 表和 MONKEYS 表中的记录。

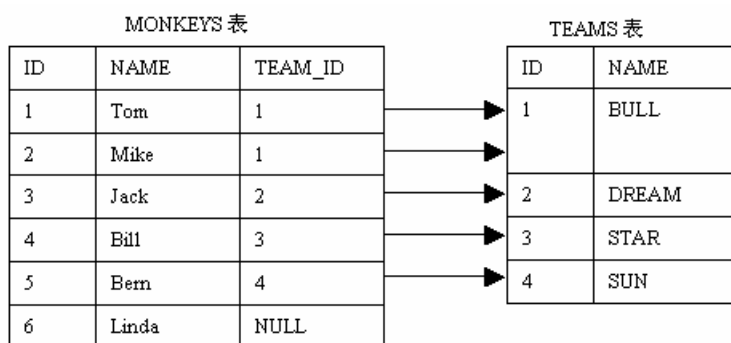


图 7-1 TEAMS 表和 MONKEYS 表中的记录

以下代码用于到数据库中检索所有的 Team 对象：

```
List teamLists=session.createQuery("from Team as t").list();
```

假定 Hibernate 加载每个 Team 对象时还会自动加载与之关联的 Monkey 对象，那么运行以上 Query 接口的 list()方法时，Hibernate 将先查询 TEAMS 表中所有的记录，然后根据每条记录的 ID，到 MONKEYS 表中查询有参照关系的记录，Hibernate 将依次执行以下 select 语句：

```
select * from TEAMS;
select * from MONKEYS where TEAM_ID=1;
select * from MONKEYS where TEAM_ID=2;
select * from MONKEYS where TEAM_ID=3;
```

```
select * from MONKEYS where TEAM_ID=4;
```

通过以上 5 条 select 语句, Hibernate 最后加载了 4 个 Team 对象和 5 个 Monkey 对象, 在内存中形成了一幅关联的对象图, 如图 7-2 所示。

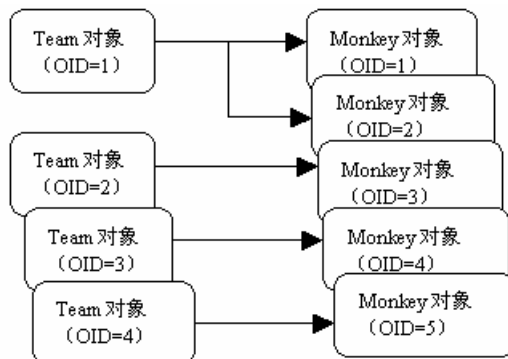


图 7-2 Team 与 Monkey 对象的关联对象图

Hibernate 检索 Team 对象时立即检索与之关联的 Monkey 对象, 这种检索策略称为立即检索策略。立即检索策略存在两大不足:

- select 语句的数目太多, 需要频繁地访问数据库, 会影响检索性能。如果需要查询  $n$  个 Team 对象, 那么必须执行  $n+1$  次 select 查询语句。这种检索策略没有利用 SQL 的连接查询功能, 例如, 以上 5 条 select 语句完全可以通过以下一条 select 语句来完成:

```
select * from TEAMS left outer join MONKEYS
on TEAMS.ID=MONKEYS.TEAM_ID
```

以上 select 语句使用了 SQL 的左外连接查询功能, 能够在一条 select 语句中查询出 TEAMS 表的所有记录, 以及有参照关系的 MONKEYS 表的记录。

- 在应用逻辑只需要访问 Team 对象, 而不需要访问 Monkey 对象的场合, 加载 Monkey 对象完全是多余的操作, 这些多余的 Monkey 对象白白浪费了许多内存空间。

为了解决以上问题, Hibernate 提供了其他两种检索策略: 延迟检索策略和迫切左外连接检索策略。延迟检索策略能避免多余加载应用程序不需要访问的关联对象, 迫切左外连接检索策略则充分利用了 SQL 的外连接查询功能, 能够减少 select 语句的数目。本章将介绍这些检索策略的运行机制和使用方法。

此外, 本章还介绍 Hibernate 为应用程序提供的 3 种检索方式:

- HQL 检索方式: 通过 Query 接口和 HQL 查询语言检索数据库的数据。
- QBC 检索方式: 通过 Criteria 等接口来检索数据库的数据。
- 本地 SQL 检索方式: 通过本地 SQL 语言来检索数据库的数据。



## 7.1 Hibernate 的检索策略

当初学者运行类似以下的程序代码时，常常会遇到 `LazyInitializationException` 异常：

```
Transaction tx = session.beginTransaction();
Team team=(Team)session.load(Team.class,new Long(1));
tx.commit();
session.close();
team.getName(); //抛出 LazyInitializationException
```

为什么加载了 `Team` 对象以后，无法访问它的 `name` 属性呢？要解决这个问题，就必须了解 `Hibernate` 的检索策略。检索策略包括立即检索、延迟检索和迫切左外连接检索，它们会影响检索方法的运行时行为。

当 `Hibernate` 通过 `session.load()` 方法加载 `Team` 对象时，需要获得以下信息：

- 类级别检索策略：`Session` 的 `load()` 方法直接指定检索的是 `Team` 对象，对 `Team` 对象到底采用立即检索，还是延迟检索？
- 关联级别检索策略：对与 `Team` 关联的 `Monkey` 对象，即 `Team` 对象的 `monkeys` 集合属性，到底采用立即检索、还是延迟检索或者迫切左外连接检索？

表 7-1 列出了这 3 种检索策略的运行机制。

表 7-1 3 种检索策略的运行机制

检索策略的类型	类 级 别	关 联 级 别
立即检索	立即加载检索方法指定的对象	立即加载与检索方法指定的对象所关联的对象
延迟检索	延迟加载检索方法指定的对象	延迟加载与检索方法指定的对象所关联的对象
迫切左外连接检索	不适用	通过左外连接加载与检索方法指定的对象所关联的对象

在对象-关系映射文件中，`<class>`、`<set>` 和 `<many-to-one>` 元素等都有一个 `lazy` 属性，`<set>` 和 `<many-to-one>` 元素还有一个 `fetch` 属性，`lazy` 属性和 `fetch` 属性的取值可决定检索策略。表 7-2 列出了 `lazy` 属性和 `fetch` 属性与检索策略的对应关系。

表 7-2 lazy 属性和 fetch 属性与检索策略的对应关系

检索策略的类型	类 级 别	关 联 级 别
立即检索	<code>&lt;class&gt;</code> 元素的 <code>lazy</code> 属性为 <code>false</code>	<code>&lt;set&gt;</code> 或 <code>&lt;many-to-one&gt;</code> 元素的 <code>lazy</code> 属性为 <code>false</code>
延迟检索	<code>&lt;class&gt;</code> 元素的 <code>lazy</code> 属性为默认值 <code>true</code>	<code>&lt;set&gt;</code> 元素的 <code>lazy</code> 属性为默认值 <code>true</code> <code>&lt;many-to-one&gt;</code> 元素的 <code>lazy</code> 属性为默认值 <code>proxy</code>
迫切左外连接检索	不适用	<code>&lt;set&gt;</code> 或 <code>&lt;many-to-one&gt;</code> 元素的 <code>fetch</code> 属性为 <code>join</code>

本书范例程序位于配套光盘的 `sourcecode\chapter7` 目录下。在运行程序之前，

需要先在 MySQL 中手工创建 SAMPLEDB 数据库、TEAMS 表和 MONKEYS 表，然后向 TEAMS 表和 MONKEYS 表中插入本章开头的图 7-1 列出的记录，相关的 SQL 脚本文件为 schema\sampladb.sql。

在 DOS 下转到本例的根目录 chapter7，输入命令：ant run，该命令将运行 BusinessService 类，BusinessService 类定义了一系列检索方法：

- loadTeam(): 用 Session 的 load()方法加载一个 Team 对象，然后通过它导航到关联的 Monkey 对象。
- getTeam(): 用 Session 的 get()方法加载一个 Team 对象，然后通过它导航到关联的 Monkey 对象。
- findAllTeams(): 用 Query 的 list()方法加载所有的 Team 对象，然后通过它们导航到关联的 Monkey 对象。
- loadMonkey(): 用 Session 的 load()方法加载一个 Monkey 对象，然后通过它导航到关联的 Team 对象。
- getMonkey(): 用 Session 的 get()方法加载一个 Monkey 对象，然后通过它导航到关联的 Team 对象。
- findAllMonkeys(): 用 Query 的 list()方法加载所有的 Monkey 对象，然后通过它们导航到关联的 Team 对象。
- findTeamLeftJoinMonkey(): 用 Query 的 list()方法加载所有的 Team 对象，并且在应用程序中指定采用迫切左外连接策略来检索关联的 Monkey 对象。

在以上每个检索方法中都会输出一些用于跟踪程序运行状态的信息。例如，以下是 loadTeam()方法的源代码：

```
tx = session.beginTransaction();

System.out.println("loadTeam():executing session.load()");
Team team=(Team)session.load(Team.class,new Long(1));

System.out.println("loadTeam():executing team.getName()");
team.getName();

System.out.println("loadTeam():executing team.getMonkeys().
    iterator()");
Iterator monkeyIterator=team.getMonkeys().iterator();

tx.commit();
```

当 Team.hbm.xml 文件的<class>元素的 lazy 属性为 false 时，loadTeam()方法向控制台输出以下信息：

```
[java] loadTeam():executing session.load()
[java] Hibernate: select team0_.ID as ID0_0_, team0_.NAME
```

```

as NAME0_0_ from TEAMS team0_ where team0_.ID=?
[java] loadTeam():executing team.getName()
[java] loadTeam():executing team.getMonkeys().iterator()
[java] Hibernate: select monkeys0_.TEAM_ID as TEAM3_1_,
monkeys0_.ID as ID1_, monkeys0_.ID as ID1_0_,
monkeys0_.NAME as NAME1_0_, monkeys0_.TEAM_ID as TEAM3_1_0_
from MONKEYS monkeys0_ where monkeys0_.TEAM_ID=?

```

当 Team.hbm.xml 文件的 <class> 元素的 lazy 属性为 true 时, loadTeam() 方法向控制台输出以下信息:

```

[java] loadTeam():executing session.load()
[java] loadTeam():executing team.getName()
[java] Hibernate: select team0_.ID as ID0_0_, team0_.NAME
as NAME0_0_ from TEAMS team0_ where team0_.ID=?
[java] loadTeam():executing team.getMonkeys().iterator()
[java] Hibernate: select monkeys0_.TEAM_ID as TEAM3_1_,
monkeys0_.ID as ID1_, monkeys0_.ID as ID1_0_,
monkeys0_.NAME as NAME1_0_, monkeys0_.TEAM_ID as TEAM3_1_0_
from MONKEYS monkeys0_ where monkeys0_.TEAM_ID=?

```

从以上输出信息可以看出, 如果 lazy 属性为 false, 在运行 session.load() 方法时 Hibernate 立即执行查询 TEAMS 表的 select 语句, 这是因为对 Team 对象采用了立即检索策略。如果 lazy 属性为默认值 true, 那么直到调用 team.getName() 方法时 Hibernate 才执行查询 TEAMS 表的 select 语句, 这是因为对 Team 对象设置了延迟检索策略。

读者还可以修改 Team.hbm.xml 和 Monkey.hbm.xml 文件中的检索策略, 然后运行 BusinessService 类, 观察在各种检索策略下生成的 select 语句, 从而加深对各种检索策略的理解。

### 7.1.1 类级别的检索策略

类级别可选的检索策略包括立即检索和延迟检索, 默认为延迟检索。如果 <class> 元素的 lazy 属性为 true, 表示采用延迟检索; 如果 lazy 属性为 false, 表示采用立即检索。lazy 属性的默认值为 true。

#### 1. 立即检索

在 Team.hbm.xml 文件中, 以下方式表示采用立即检索策略:

```
<class name="mypack.Team" table="TEAMS" lazy="false">
```

当通过 Session 的 load() 方法检索 Team 对象时:

```
Team team=(Team)session.load(Team.class,new Long(1));
```

Hibernate 会立即执行查询 TEAMS 表的 select 语句:

```
select * from TEAMS where ID=1;
```

## 2. 延迟检索

类级别的默认检索策略为延迟检索。在 Team.hbm.xml 文件中，以下两种方式都表示采用延迟检索策略：

```
<class name="mypack.Team" table="TEAMS" >
或者
<class name="mypack.Team" table="TEAMS" lazy="true">
```

当执行 Session 的 load(Team.class,new Long(1))方法时，Hibernate 不会立即执行查询 TEAMS 表的 select 语句，仅返回 Team 类的代理类的实例，这个代理类具有以下特征：

- 由 Hibernate 在运行时动态生成，它扩展了 Team 类，因此它继承了 Team 类的所有属性和方法，但它的实现对于应用程序是透明的。
- 当 Hibernate 创建 Team 代理类实例时，仅初始化了它的 OID 属性，其他属性都为 null，因此这个代理类实例占用的内存很少。
- 当应用程序第一次访问 Team 代理类实例时（例如调用 team.getXXX()或 team.setXXX()方法），Hibernate 会初始化代理类实例，在初始化过程中执行 select 语句，真正从数据库中加载 Team 对象的所有数据。但有个例外，那就是当应用程序访问 Team 代理类实例的 getId()方法时，Hibernate 不会初始化代理类实例，因为在创建代理类实例时 OID 就存在了，不必到数据库中去查询。

以下代码先通过 Session 的 load()方法加载 Team 对象，然后访问它的 name 属性：

```
tx = session.beginTransaction();
Team team=(Team)session.load(Team.class,new Long(1)); //返回 Team 代理类实例
team.getName(); //初始化 Team 代理类实例，执行 select 语句
tx.commit();
```

当运行 session.load()方法时，Hibernate 不执行任何 select 语句，仅返回 Team 类的代理类的实例，它的 OID 为 1，这是由 load()方法的第二个参数指定的。当应用程序调用 team.getName()方法时，Hibernate 会初始化 Team 代理类实例，从数据库中加载 Team 对象的数据，执行以下 select 语句：

```
select * from TEAMS where ID=1;
```

当<class>元素的 lazy 属性为 true 时，会影响 Session 的 load()方法的各种运行时行为，下面举例说明。

(1) 如果加载的 Team 对象在数据库中不存在，Session 的 load()方法不会抛出异常，只有当运行 team.getName()方法时才会抛出以下异常：

```
Exception in thread "main" org.hibernate.ObjectNotFoundException:
```

No row with the given identifier exists

(2) 如果在整个 Session 范围内, 应用程序没有访问过 Team 对象, 那么 Team 代理类的实例一直不会被初始化, Hibernate 不会执行任何 select 语句。以下代码试图在关闭 Session 后访问 Team 游离对象:

```
tx = session.beginTransaction();
Team team=(Team)session.load(Team.class,new Long(1));
tx.commit();
session.close();
String name=team.getName(); //抛出LazyInitializationException
```

由于引用变量 team 引用的 Team 代理类的实例在 Session 范围内始终没有被初始化, 因此在执行 team.getName()方法时, Hibernate 会抛出以下异常:

```
Exception in thread "main":org.hibernate.LazyInitializationException
could not initialize proxy - no Session
```

由此可见, Team 代理类的实例只有在当前 Session 范围内才能被初始化。

(3) org.hibernate.Hibernate 类的 initialize()静态方法用于在 Session 范围内显式初始化代理类实例, isInitialized()方法用于判断代理类实例是否已经被初始化。例如:

```
tx = session.beginTransaction();
Team team=(Team)session.load(Team.class,new Long(1));
.....
if(!Hibernate.isInitialized(team))
    Hibernate.initialize(team);
tx.commit();
session.close();
String name=team.getName(); //正常执行
```

以上代码在 Session 范围内通过 Hibernate 类的 initialize()方法显式初始化了 Team 代理类实例, 因此当 Session 关闭后, 可以正常访问 Team 游离对象。

(4) 当应用程序访问代理类实例的 getId()方法时, 不会触发 Hibernate 初始化代理类实例的行为, 例如:

```
tx = session.beginTransaction();
Team team=(Team)session.load(Team.class,new Long(1));
team.getId();
tx.commit();
session.close();
String name=team.getName(); //抛出异常
```

当应用程序访问 team.getId()方法时, 该方法直接返回 Team 代理类实例的 OID 值, 无须查询数据库。由于引用变量 team 始终引用的是没有被初始化的 Team 代理类实例,

因此当 Session 关闭后再执行 team.getName()方法，Hibernate 会抛出以下异常：

```
Exception in thread "main" org.hibernate.LazyInitializationException:
could not initialize proxy - no Session
```

值得注意的是，不管 Team.hbm.xml 文件的<class>元素的 lazy 属性是 true 还是 false，Session 的 get()方法及 Query 的 list()方法在 Team 类级别总是使用立即检索策略，下面举例说明。

(1) Session 的 get()方法总是立即到数据库中检索 Team 对象，如果在数据库中不存在相应的数据，就返回 null。例如：

```
Team team=(Team)session.get(Team.class,new Long(1));
```

当通过 Session 的 get()方法加载 Team 对象时，Hibernate 会立即执行以下 select 语句：

```
select * from TEAMS where ID=1;
```

如果存在相关的数据，get()方法就返回 Team 对象，否则就返回 null。get()方法永远不会返回 Team 代理类实例，这是与 load()方法的不同之处。

(2) Query 的 list()方法总是立即到数据库中检索 Team 对象，例如：

```
List teamLists=session.createQuery("from Team as t").list();
```

当运行 Query 的 list()方法时，Hibernate 立即执行以下 select 语句：

```
select * from TEAMS;
```

### 7.1.2 一对多和多对多关联的检索策略

在映射文件中，用<set>元素来配置一对多关联及多对多关联关系。Team.hbm.xml 文件中的以下代码用于配置 Team 和 Monkey 类的一对多关联关系：

```
<set name="monkeys" inverse="true" >
    <key column="TEAM_ID" />
    <one-to-many class="mypack.Monkey" />
</set>
```

<set>元素有 lazy 和 fetch 属性，表 7-3 列出了这两个属性取不同值时对 Team 类的 monkeys 集合采用的检索策略。

表 7-3 <set>元素的 lazy 和 fetch 属性

lazy 属性	fetch 属性	检索策略
true (默认值)	未显式设置	采用延迟检索，这是默认的检索策略。这是在一对多或多对多关联级别优先考虑使用的检索策略
false	未显式设置	采用立即检索
未显式设置	join	采用迫切左外连接检索

假如没有显式设置<set>元素的 lazy 和 fetch 属性,那么采用默认的延迟检索策略。例如以下代码表明对 monkeys 集合采用延迟检索:

```
<set name="monkeys" inverse="true" >.....</set>
```

如果把 fetch 属性设为 join,那么就表示采用迫切左外连接检索, lazy 属性被忽略,此时显式设置 lazy 属性是无意义的:

```
<set name="monkeys" inverse="true" fetch="join">.....</set>
```

### Tips

<set>元素不仅用于映射一对多和多对多关联,还能映射存放值类型数据的集合,参见第 12 章(映射值类型集合)。本节主要以 Team 和 Monkey 类的一对多关联为例,介绍如何在<set>元素中设置检索策略,这其实也适用于多对多关联和存放值类型数据的集合。

在 Team 类中定义了一个 java.util.Set 集合类型的 monkeys 属性:

```
private Set monkeys=new HashSet();
public Set getMonkeys() {
    return this.monkeys;
}
public void setMonkeys(Set monkeys) {
    this.monkeys = monkeys;
}
```

Hibernate 为 Set 集合类也提供了代理类,它扩展了 Set 接口,但它的实现应用程序是透明的。与持久化类的代理类不同的是,不管有没有设置延迟检索策略, Hibernate 的各种检索方法在为 Team 对象的 monkeys 属性赋值时, monkeys 属性总是引用集合代理类的实例。关于集合代理类的作用可参见本书第 12 章的 12.6 节(小结)。

### 1. 立即检索 (lazy 属性为 “false”)

以下代码表明对 Team 的 monkeys 集合采用立即检索:

```
<set name="monkeys" inverse="true" lazy="false" >.....</set>
```

以下代码通过 Session 的 get()方法加载 OID 为 1 的 Team 对象:

```
tx = session.beginTransaction();
Team team=(Team)session.get(Team.class,new Long(1));
Set monkeys= team.getMonkeys();
Iterator monkeyIterator=monkeys.iterator();
tx.commit();
```

执行 Session 的 get()方法时,对 Team 对象采用类级别的立即检索策略;对 Team 对象的 monkeys 集合(即与 Team 关联的所有 Monkey 对象)采用一对多关联级别的立即检索策略,因此 Hibernate 执行以下 select 语句:

```
select * from TEAMS where ID=1;
```

```
select * from MONKEYS where TEAM_ID=1;
```

通过以上 select 语句，Hibernate 加载了一个 Team 对象和两个 Monkey 对象。Team 对象的 monkeys 属性引用的是一个 Hibernate 提供的 Set 代理类实例，这个代理类实例引用两个 Monkey 对象，参见图 7-3。

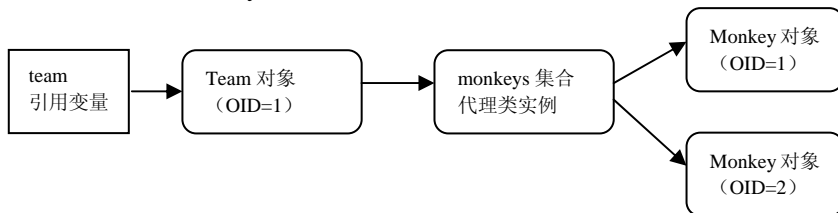


图 7-3 Team 对象的 monkeys 属性引用一个集合代理类实例

假如一个武术队有 100 个猴子，Session 的 get() 方法会立即加载一个 Team 对象和 100 个 Monkey 对象，但在很多情况下，应用程序并不需要访问这些 Monkey 对象，所以在一对多关联级别中不能随意使用立即检索策略。

## 2. 延迟检索（lazy 属性为默认值 “true”）

对于 <set> 元素，应该优先考虑使用默认的延迟检索策略：

```
<set name="monkeys" inverse="true">.....</set>
或者
<set name="monkeys" inverse="true" lazy="true">.....</set>
```

此时运行 Session 的 get(Team.class, new Long(1)) 方法时，仅立即检索 Team 对象，执行以下 select 语句：

```
select * from TEAMS where ID=1;
```

值得注意的是，尽管 Hibernate 会延迟检索与 Team 对象关联的 Monkey 对象，但没有创建 Monkey 代理类实例。事实上，这时也无法创建 Monkey 代理类实例，因为无法知道与 Team 关联的所有 Monkey 对象的 OID。get() 方法返回的 Team 对象的 monkeys 属性引用的是 Hibernate 提供的集合代理类实例。只有当 monkeys 集合代理类实例被初始化时，才会到数据库中检索所有与 Team 关联的 Monkey 对象，执行以下 select 语句：

```
select * from MONKEYS where TEAM_ID=1;
```

那么，monkeys 集合代理类实例什么时候被初始化呢？主要包括以下两种情况：

（1）当应用程序第一次访问它，如调用它的 iterator()、size()、isEmpty() 或 contains() 方法：

```
Set monkeys=team.getMonkeys();

//导致 monkeys 集合代理类实例被初始化，执行查询 MONKEYS 表的 select 语句
```



```
Iterator it=monkeys.iterator();
```

(2) 通过 `org.hibernate.Hibernate` 类的 `initialize()` 静态方法初始化它:

```
Set monkeys=team.getMonkeys();
```

```
//导致 monkeys 集合代理类实例被初始化, 执行查询 MONKEYS 表的 select 语句
Hibernate.initialzie(monkeys());
```

### 3. 迫切左外连接检索 (fetch 属性为 “join”)

如果把 `<set>` 元素的 `fetch` 属性设为 `join`:

```
<set name="monkeys" inverse="true" fetch="join" >
```

那么当检索 `Team` 对象时, 会采用迫切左外连接检索策略来检索所有关联的 `Monkey` 对象。对于以下程序:

```
Team team=(Team)session.get(Team.class,new Long(1));
```

当运行 `Session` 的 `get()` 方法时, 执行以下 `select` 语句:

```
select * from TEAMS left outer join MONKEYS
on TEAMS.ID =MONKEYS.TEAM_ID where TEAMS.ID=1;
```

值得注意的是, `Query` 的 `list()` 方法会忽略映射文件中配置的迫切左外连接检索策略。即使以上 `<set>` 元素的 `fetch` 属性为 `join`, 对于以下代码:

```
List teamLists=session.createQuery("from Team as t").list();
```

Hibernate 对 `Team` 对象的 `monkeys` 集合仍然采用延迟检索策略, Hibernate 执行的 `select` 语句为:

```
select * from TEAMS;
```

只有当程序访问 `monkeys` 集合时, 才会通过相应的 `select` 语句初始化 `monkeys` 集合代理类实例。

#### 7.1.3 多对一和一对一关联的检索策略

在映射文件中, `<many-to-one>` 及 `<one-to-one>` 元素分别用来设置多对一和一对一关联关系。在 `Monkey.hbm.xml` 文件中, 以下代码设置 `Monkey` 类与 `Team` 类的多对一关联关系。

```
<many-to-one name="team" column="TEAM_ID" class="mypack.Team"/>
```

和 `<set>` 元素一样, `<many-to-one>` 元素也有一个 `lazy` 属性和 `fetch` 属性。表 7-4 列出了 `<many-to-one>` 元素的 `lazy` 属性及 `fetch` 属性取不同值时设置的检索策略。

表 7-4 设置多对一关联的检索策略

lazy 属性	fetch 属性	检索 Monkey 对象时 对关联的 Team 对象使用的检索策略
proxy (默认值)	未显式设置	延迟检索
false	未显式设置	立即检索
未显式设置	join	迫切左外连接检索

在默认情况下，在多对一和一对一关联级别使用延迟检索策略。如果把 fetch 属性设为 join，那么 lazy 属性被忽略，此时显式设置 lazy 属性是无意义的。

对于多对一或一对一关联，使用外连接检索策略的优点在于比立即检索策略使用的 select 语句数目少。不过，假如应用程序仅希望访问 Monkey 对象，并不需要立即访问与 Monkey 关联的 Team 对象，则可以使用延迟检索策略。

### 1. 立即检索 (lazy 属性为 “false”)

以下代码把 Monkey.hbm.xml 文件的 <many-to-one> 元素的 lazy 属性设为 false:

```
<many-to-one
    name="team"
    column="TEAM_ID"
    class="mypack.Team"
    lazy="false"
/>
```

对于以下程序代码:

```
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
```

当运行 session.get() 方法时，Hibernate 执行以下 select 语句:

```
select * from MONKEYS where ID=1;
select * from TEAMS where ID=1;
```

### 2. 延迟检索 (lazy 属性为默认值 “proxy”)

如果希望检索 Monkey 对象时，延迟检索关联的 Team 对象，只要让 Monkey.hbm.xml 文件中的 <many-to-one> 元素的 lazy 属性为默认值 proxy。

对于以下程序代码:

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
Team team=monkey.getTeam();
String name=team.getName(); //初始化 Team 代理类实例
tx.commit();
```

当运行 Session 的 get() 方法时，仅立即执行检索 Monkey 对象的 select 语句:

```
select * from MONKEYS where ID=1;
```

Monkey 对象的 team 属性引用 Team 代理类实例，这个代理类实例的 OID 由 MONKEYS 表的 TEAM\_ID 外键值决定。monkey.getTeam()方法返回 Team 代理类实例的引用，参见图 7-4。

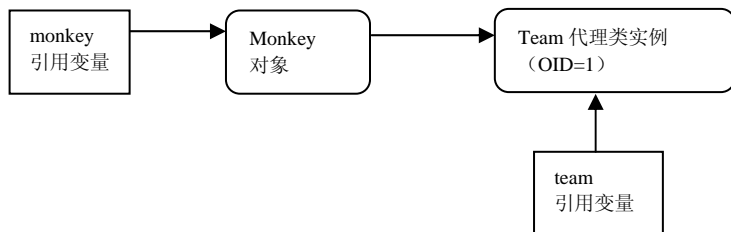


图 7-4 Monkey 对象与 Team 代理类的实例关联

在图 7-4 中，team 引用变量引用 Team 代理类实例，当执行 team.getName()方法时，Hibernate 初始化 Team 代理类实例，执行以下 select 语句，以便到数据库中加载 Team 对象的数据：

```
select * from TEAMS where ID=1;
```

对于一对一关联，如果使用延迟加载策略，必须把<one-to-one>元素的 constrained 属性设为 true：

```
<one-to-one name="team" class="mypack.Team" constrained="true" />
```

<one-to-one>元素的 constrained 属性与<many-to-one>元素的 not-null 属性在语义上有些相似，它表明 Monkey 对象必须和一个 Team 对象关联，即 Monkey 对象的 team 属性不允许为 null。

### 3. 迫切左外连接检索（fetch属性为“join”）

如果 Monkey.hbm.xml 文件中<many-to-one>元素的 fetch 属性为 join，那么在检索 Monkey 对象时，对关联的 Team 对象使用迫切左外连接检索策略。

对于以下程序代码：

```
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
```

在运行 session.get()方法时，Hibernate 需要决定以下检索策略：

- 类级别的 Monkey 对象的检索策略：根据本章 7.1.1 节的介绍，get()方法在类级别总是使用立即检索策略。
- 与 Monkey 多对一关联的 Team 对象（即 Monkey 对象的 team 属性）的检索策略：假定 Monkey.hbm.xml 文件中<many-to-one>元素的 fetch 属性为 join，因此使用迫切左外连接检索策略。
- 与 Team 一对多关联的 Monkey 对象的检索策略：假定 Team.hbm.xml 文件中<set>元素的 lazy 属性为 false，因此使用立即检索策略。

根据以上检索策略，Hibernate 执行以下 select 语句：

```
select * from MONKEYS left outer join TEAMS
on MONKEYS.TEAM_ID=TEAMS.ID where MONKEYS.ID=1

select * from MONKEYS where TEAM_ID=1
```

通过以上两条 select 语句，Session 的 get() 方法实际上加载了 3 个持久化对象，如图 7-5 所示。

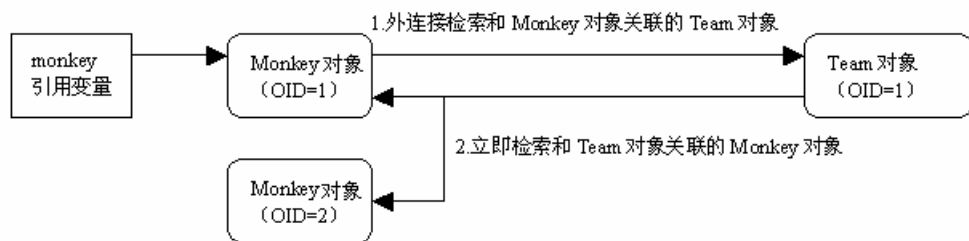


图 7-5 Session 的 get() 方法加载的 3 个持久化对象的对象图

假定 Team.hbm.xml 文件中 <set> 元素的 lazy 属性为 true，那么对与 Team 关联的 Monkey 对象采用延迟检索策略，Hibernate 仅执行以下 select 语句：

```
select * from MONKEYS left outer join TEAMS
on MONKEYS.TEAM_ID=TEAMS.ID where MONKEYS.ID=1;
```

通过以上一条 select 语句，Session 的 get() 方法实际上加载了两个持久化对象，如图 7-6 所示。

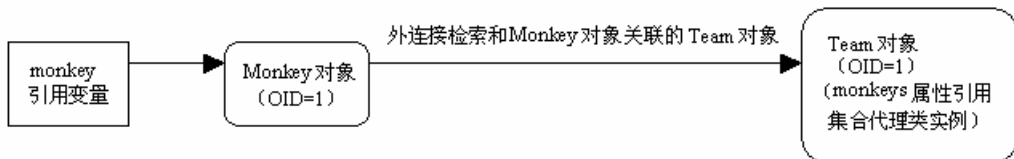


图 7-6 Session 的 get() 方法加载的两个持久化对象的对象图

值得注意的是，Query 的 list() 方法会忽略映射文件中配置的迫切左外连接检索策略。假定在 Monkey.hbm.xml 文件中 <many-to-one> 元素的 fetch 属性为 join，对于以下代码：

```
List monkeyLists=session.createQuery("from Monkey as m ").list();//第1行
Iterator monkeyIterator=monkeyLists.iterator();//第2行
Monkey monkey1=(Monkey)monkeyIterator.next();//第3行
Team team1=monkey1.getTeam();//第4行
String name=team1.getName();//第5行
```

Hibernate 对与 Monkey 关联的 Team 对象仍然采用延迟检索策略。在程序第一行, Hibernate 执行的 select 语句为:

```
select * from MONKEYS;
```

程序第一行加载的所有 Monkey 对象的 team 属性都引用 Team 代理类实例。在程序第 5 行, 会初始化一个 Team 代理类实例, 执行以下 select 语句:

```
select * from TEAMS where ID=1;
```

#### 7.1.4 在应用程序中显式指定迫切左外连接检索策略

在映射文件中设定的检索策略是固定的, 要么为延迟检索, 要么为立即检索, 要么为迫切左外连接检索。但应用逻辑是多种多样的, 有些情况下需要延迟检索, 而有些情况下需要迫切左外连接检索。Hibernate 允许在应用程序中覆盖映射文件中设定的检索策略, 由应用程序在运行时决定检索对象图的深度。

以下 Query 的 list() 方法都用于检索 name 属性以 “B” 开头的 Team 对象:

```
session.createQuery("from Team as t where t.name like 'B%'").list();

session
    .createQuery("from Team as t left join fetch t.monkeys "
        + "where t.name like 'B%' ")
    .list();
```

在执行第一个 Query 的 list() 方法时, 将使用映射文件配置的检索策略。在执行第二个 Query 的 list() 方法时, 在 HQL 语句中显式指定迫切左外连接检索关联的 Monkey 对象, 因此会覆盖映射文件配置的检索策略, 不管在 Team.hbm.xml 文件中 <set> 元素的 lazy 属性是 true 还是 false, Hibernate 都会执行以下 select 语句:

```
select * from TEAMS left outer join MONKEYS
on TEAMS.ID =MONKEYS.TEAM_ID where TEAMS.NAME like 'B%';
```

如果采用 QBC 检索方式, 那么静态常量 FetchMode.JOIN 表示采用迫切左外连接检索策略:

```
List result=session
    .createCriteria(Team.class)
    .add(Restrictions.like("name", "B%"))
    //对 Team 的 monkeys 集合采用迫切左外连接检索
    .setFetchMode("monkeys", FetchMode.JOIN)
    .list();
```

#### 7.1.5 比较 3 种检索策略

表 7-5 总结了这几种策略的优缺点, 以及各自优先考虑使用的场合。

表 7-5 比较 Hibernate 的 3 种检索策略

检索策略	优点	缺点	优先考虑使用的场合
立即检索	对应用程序完全透明，不管对象处于持久化状态，还是游离状态，应用程序都可以方便地从一个个对象导航到与它关联的对象	(1) select 语句数目多 (2) 可能会加载应用程序不需要访问的对象，白白浪费许多内存空间	(1) 类级别 (2) 应用程序需要立即访问的 (3) 使用了 Hibernate 的第二级缓存
延迟检索	由应用程序决定需要加载哪些对象，可以避免执行多余的 select 语句，以及避免加载应用程序离状态的代理类实例，必须不需要访问的对象。因此能提高检索性能，并且保证它在持久化状态时已经或者根本不会访问的对象能节省内存空间	应用程序如果希望访问游离状态的代理类实例，必须被初始化	(1) 一对多或者多对多关联 (2) 应用程序不需要立即访问
迫切左外连接检索	(1) 对应用程序完全透明，不管对象处于持久化状态，还是游离状态，应用程序都可以方便地从一个个对象导航到与它关联的对象 (2) 使用了外连接，select 语句数目少	(1) 可能会加载应用程序不需要访问的对象，白白浪费许多内存空间 (2) 复杂的数据库表连接也会影响检索性能	(1) 多对一或者一对一关联 (2) 应用程序需要立即访问的对象 (3) 数据库系统具有良好的表连接性能

对于立即检索和延迟检索策略，在查询每张表时都使用单独的 select 语句，这种查询方式的优点在于每个 select 语句很简单，查询速度快，缺点在于 select 语句的数目多，增加了访问数据库的频率。迫切左外连接检索运用了 SQL 外连接的查询功能，优点在于 select 语句的数目少，能够减少访问数据库的频率，缺点在于 select 语句复杂度提高了，数据库系统建立表之间的连接也是耗时的操作。

在映射文件中配置的检索策略是固定的，Hibernate 还允许在应用程序的 HQL 语句中显式指定迫切左外连接检索策略，它会覆盖映射文件中配置的检索策略。

对于实际的应用，为了选择合适的检索策略，需要测试应用程序的各个用例，跟踪使用不同检索策略时 Hibernate 执行的 SQL 语句。可以把 Hibernate 配置文件的 show\_sql 属性设为 true，使得 Hibernate 在运行时输出执行的 SQL 语句。根据特定的关系模型，评估各种查询语句的性能，比较到底是使用外连接查询速度快：

```
select * from TEAMS left outer join MONKEYS
on TEAMS.ID =MONKEYS.TEAM_ID where TEAMS.ID=1;
```

还是使用分开的 select 语句速度更快：

```
select * from TEAMS where ID=1;
select * from MONKEYS where TEAM_ID=1;
```

不断地调节检索策略，以便在减少 select 语句数目和减少 select 语句复杂度之间找到一个平衡点，获得最佳的检索性能。

## 7.2 检索方式

Hibernate 提供了 3 种检索方式：HQL 检索方式、QBC 检索方式和本地 SQL 检索方式，下面介绍这些检索方式的特点及使用场合。

### 7.2.1 HQL检索方式

HQL（Hibernate Query Language）是面向对象的查询语言，它和 SQL 查询语言在形式上有些相似。在 Hibernate 提供的各种检索方式中，HQL 是使用最广的一种检索方式。以下程序代码用于检索姓名为“Tom”，并且年龄为 21 的 Monkey 对象：

```
//创建一个 Query 对象
Query query=session.createQuery("from Monkey as m where m.name=
    :monkeyName "+"and m.age=:monkeyAge");
//动态绑定参数
query.setString("monkeyName","Tom");
query.setInteger("monkeyAge",21);

//执行查询语句，返回查询结果
List result= query.list();
```

从以上程序代码看出，HQL 检索方式包括以下步骤。

（1）通过 Session 的 createQuery()方法创建一个 Query 对象，它包含一个 HQL 查询语句。HQL 查询语句可以包含命名参数，如“monkeyName”和“monkeyAge”都是命名参数。

（2）动态绑定参数。Query 接口提供了给各种类型的命名参数赋值的方法，例如 setString()方法用于为字符串类型的 monkeyName 命名参数赋值。

（3）调用 Query 的 list()方法执行查询语句。该方法返回 List 类型的查询结果，在 List 集合中存放了符合查询条件的持久化对象。对于以上程序代码，当运行 Query 的 list()方法时，Hibernate 执行以下 SQL 查询语句：

```
select * from MONKEYS where NAME='Tom' and AGE=21;
```

Query 接口支持方法链编程风格，它的 setString()方法及其他 setXXX()方法都返回自身实例，而不是返回 void 类型。以下是 Query 接口的实现类中 setString()方法的源程序：

```
public Query setString(int position, String val) {
    setParameter(position, val, Hibernate.STRING);
    return this;
}
```

如果采用方法链编程风格，将按以下形式访问 Query 接口：

```
List result=session.createQuery(".....")
    .setString("monkeyName","Tom")
    .setInteger("monkeyAge",21)
    .list();
```

可见，方法链编程风格能使程序代码更加简洁。

尽管 HQL 与 SQL 在语法形式上有些相似，例如：

```
HQL 查询语句: from Monkey as m where m.name='Tom' and m.age=21
SQL 查询语句: select * from MONKEYS where NAME='Tom' and AGE=21;
```

但 HQL 与 SQL 在本质上是不同的：

- HQL 查询语句是面向对象的，由 Hibernate 对其解析，然后根据对象-关系的映射信息来翻译成相应的 SQL 语句。HQL 查询语句中的主体是域模型中的类及类的属性。例如在以上 HQL 查询语句例子中，“Monkey”是持久化类的名字，“m.age”是持久化类的属性的名字。
- SQL 查询语句是与关系数据库绑定在一起的。SQL 查询语句中的主体是数据库表及表的字段。例如在以上 SQL 查询语句例子中，“MONKEYS”是表的名字，“AGE”是表的字段的名称。

## 7.2.2 QBC检索方式

采用 HQL 检索方式时，在应用程序中需要定义基于字符串形式的 HQL 查询语句。QBC API 提供了检索对象的另一种方式，它主要由 org.hibernate.Criteria 接口、org.hibernate.criterion.Criterion 接口和 org.hibernate.criterion.Restrictions 类组成，它支持在运行时动态生成查询语句。

以下程序代码用于检索姓名以字符“T”开头，并且年龄为 21 的 Monkey 对象：

```
//创建一个 Criteria 对象
Criteria criteria=session.createCriteria(Monkey.class);

//设定查询条件，然后把查询条件加入到 Criteria 中
Criterion criterion1= Restrictions.like("name", "T%");
Criterion criterion2= Restrictions.eq("age", new Integer(21));

criteria=criteria.add(criterion1);
criteria=criteria.add(criterion2);

//执行查询语句，返回查询结果
List result=criteria.list();
```

从以上程序代码看出，QBC 检索方式包括以下步骤。



(1) 调用 Session 的 createCriteria()方法创建一个 Criteria 对象。

(2) 设定查询条件。Restrictions 类提供了一系列用于设定查询条件的静态方法，这些静态方法都返回 Criterion 实例，每个 Criterion 实例代表一个查询条件。Criteria 的 add()方法用于加入查询条件。

(3) 调用 Criteria 的 list()方法执行查询语句。该方法返回 List 类型的查询结果，在 List 集合中存放了符合查询条件的持久化对象。对于以上程序代码，当运行 Criteria 的 list()方法时，Hibernate 执行的 SQL 查询语句为：

```
select * from MONKEYS where NAME like 'T%' and AGE=21;
```

Criteria 接口支持方法链编程风格，它的 add()方法返回自身实例，而不是返回 void 类型。以下是 Criteria 接口的实现类中 add()方法的源程序：

```
public Criteria add(Criterion expression) {
    CriteriaImpl.this.add(rootAlias, expression);
    return this;
}
```

如果采用方法链编程风格，将按以下形式访问 Criteria 接口：

```
List result=session.createCriteria(Monkey.class)
    .add(Restrictions.like("name", "T%")
    .add(Restrictions.eq("age", new Integer(21))
    .list();
```

### 7.2.3 SQL检索方式

采用 HQL 或 QBC 检索方式时，Hibernate 会生成标准的 SQL 查询语句，适用于所有的数据库平台，因此这两种检索方式都是跨平台的。

有的应用程序可能需要根据底层数据库的 SQL 方言，来生成一些特殊的查询语句。在这种情况下，可以利用 Hibernate 提供的 SQL 检索方式。以下程序代码用于检索姓名以字符“T”开头，并且年龄为 21 的 Monkey 对象：

```
//创建 Query 对象
Query query=session.createSQLQuery(
    "select * from MONKEYS where NAME like :monkeyName "
    +"and AGE=:monkeyAge");

//动态绑定参数
query.setString("monkeyName", "T%");
query.setInteger("monkeyAge", 21);

//执行 SQL select 语句，返回查询结果
List result=query.list();
```

从以上程序代码看出，SQL 检索方式与 HQL 检索方式都使用 Query 接口，区别在于 SQL 检索方式通过 Session 的 createSQLQuery()方法来创建 SQLQuery 对象（SQLQuery 接口继承了 Query 接口），这个方法的参数指定一个 SQL 查询语句，该语句可以使用本地数据库的 SQL 方言。

## 7.3 小结

大致说来，Hibernate 提供了 3 种检索策略：

- 立即检索：通过 select 语句立即检索对象。
- 延迟检索：当程序检索某个对象时，仅生成代理类实例，只有当程序真正访问对象时，才会初始化代理类实例，到数据库中加载相应的数据。
- 迫切左外连接检索策略：通过左外连接查询语句立即检索出对象。

在<class>、<set>和<many-to-one>元素中都有 lazy 属性，这些元素的 lazy 属性的默认值分别为：true、true 和 proxy。由此可见，在类级别和关联级别，默认情况下都采用延迟检索策略。值得注意的是，在类级别，<class>元素的 lazy 属性仅决定 Session 的 load()方法使用的检索策略，而 Session 的 get()方法及 Query 接口的 list()方法会忽略 lazy 属性，在类级别总是使用立即检索策略。

本章还介绍了 Hibernate 提供的 HQL、QBC 及本地 SQL 检索方式的用法。HQL 的检索功能最强大，它的查询语句和 SQL 查询语句比较相似，具有较好的可读性。QBC 适合于生成动态查询语句，本地 SQL 检索方式适合于利用数据库的本地方言来生成查询语句的场合。

## 第 8 章 映射组成关系

在第 5 章，悟空已经掌握了对关联关系进行映射的方法。在本章，悟空将进一步了解如何映射组成关系。

在对象模型中，有些类由几个部分类组成，部分类的对象的生命周期依赖于整体类的对象的生命周期，当整体消失时，部分也就随之消失。这种整体与部分的关系被称为聚集关系。例如，计算机系统就是一个聚集体，它由主机箱（CpuBox）、键盘（Keyboard）、鼠标（Mouse）、显示器（Monitor）和打印机（Printer）等组成，还可能包括几个音箱（Speaker）。而主机箱（CpuBox）内除 CPU 外，还包含一些驱动设备，如显示卡（Graphics Card）和声卡（Sound Card）等。图 8-1 显示了计算机系统的组成，整体类（如计算机系统）位于层次结构的顶部，以下依次是各个部分类，每个部分类还可以由其他部分类组成。

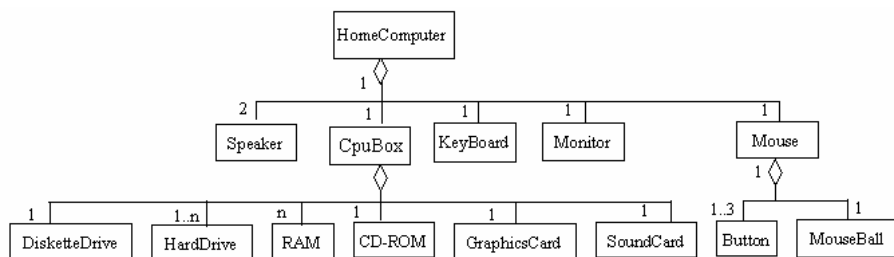


图 8-1 具有聚集关系的计算机系统

在有些情况下，部分类的对象可以被多个整体类的对象共享，例如在家庭影院系统中，电视机和录像机可以共用一个遥控器，那么这个遥控器既是电视机的组成部分，也是录像机的组成部分。还有一些情况下，一个部分类的对象只能属于一个整体类的特定对象，而不能被同一个整体类的其他对象或者被其他整体类的对象共享，例如，人和手是整体与部分的关系，每双手只能属于特定的人，张三的手永远不可能变成李四的手，更不可能变成黑猩猩的手。如果部分只能属于特定的整体，这种聚集关系也称为组成关系，在 UML 中，用实心菱形箭头表示组成关系，参见图 8-2。

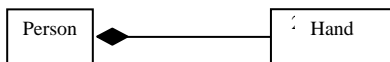


图 8-2 人和手的组成关系

本章以 Monkey 类和 Address 类的关系，以及 Computer 类、CpuBox 类、GraphicsCard 类和 Vendor 类的关系为例，介绍如何映射组成关系。本章的

BusinessService 类用于演示如何保存、更新、删除或查询具有组成关系的 Java 对象。

## 8.1 建立精粒度对象模型

假定在 Monkey 类中有以下代表家庭地址及工作地址的属性：

```
private String homeProvince;    //家庭地址所在的省
private String homeCity;       //家庭地址所在的城市
private String homeStreet;     //家庭地址所在的街道
private String homeZipcode;    //家庭地址的邮编
private String comProvince;    //工作地址所在的省
private String comCity;        //工作地址所在的城市
private String comStreet;      //工作地址所在的街道
private String comZipcode;     //工作地址的邮编
```

为了提高程序代码的可重用性，不妨从 Monkey 类中抽象出单独的 Address 类，不仅 Monkey 类可以引用 Address 类，如果日后又增加了 Gorilla 类，它也包含地址信息，那么 Gorilla 类也能引用 Address 类。按这种设计思想创建的对象模型称为精粒度对象模型，参见图 8-3，它可以最大程度地提高代码的重用性。Monkey 类与 Address 类之间为组成关系，因为它们的关系有以下特征：

- Address 对象的生命周期依赖于 Monkey 对象。当删除一个 Monkey 对象时，应该把相关的 Address 对象删除。
- 一个 Address 对象只能属于某个特定的 Monkey 对象，不能被其他 Monkey 对象共享。

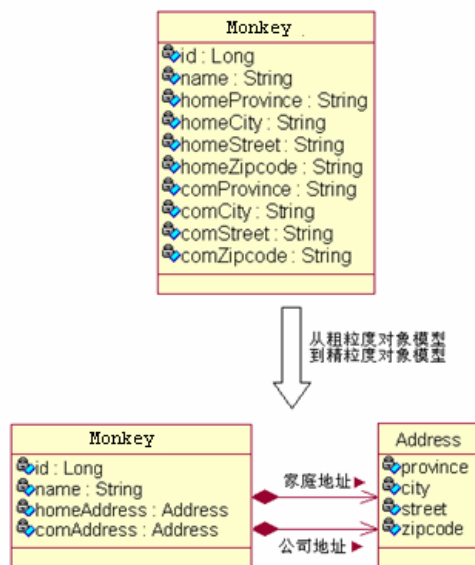


图 8-3 从粗粒度对象模型到精粒度对象模型

在精粒度对象模型中，只需为 **Monkey** 类定义两个 **Address** 类型的属性，来存放家庭地址和工作地址信息：

```
private Address homeAddress;
private Address comAddress;
```

**Monkey** 类和 **Address** 类的源代码参见例程 8-1 和例程 8-2。

例程 8-1 Monkey.java

```
package mypack;
public class Monkey{
    private Long id;
    private String name;
    private mypack.Address homeAddress;
    private mypack.Address comAddress;

    //此处省略构造方法，以及所有属性的访问方法
    .....
}
```

例程 8-2 Address.java

```
package mypack;
public class Address {
    private String street;
    private String city;
    private String province;
    private String zipcode;
    private Monkey monkey;

    //此处省略构造方法，以及所有属性的访问方法
    .....
}
```

## 8.2 建立粗粒度关系数据模型

建立关系数据模型的一个重要原则是在不会导致数据冗余的前提下，尽可能减少数据库表的数目及表之间的外键参照关系。因为如果表之间的外键参照关系很复杂，那么数据库系统在每次对关系数据进行插入、更新、删除和查询等 **SQL** 操作时，都必须建立多个表的连接，这是很耗时的操作，会影响数据库的运行性能。

以 **MONKEYS** 表为例，一种方案是把猴子的地址信息放在单独的 **ADDRESS** 表中，然后建立两个表之间的外键参照关系，如图 8-4 所示。

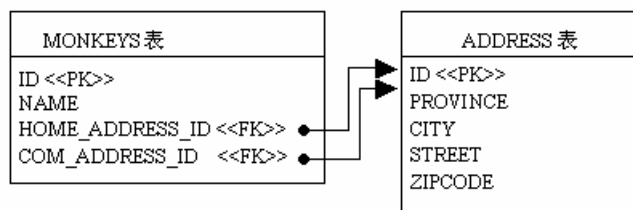


图 8-4 MONKEYS 表参照 ADDRESS 表

这使得每次查询猴子信息时，都要建立这两个表的连接。例如，以下 SQL 语句用于查询名为“Tom”的猴子的家庭地址：

```
select PROVINCE,CITY,STREET,ZIPCODE from MONKEYS as m, ADDRESS as a
where m.HOME_ADDRESS_ID =a.ID and m.NAME='Tom';
```

建立表的连接是很耗时的操作，为了提高数据库运行性能，没有必要拆分 MONKEYS 表和 ADDRESS 表，只需要在 MONKEYS 表中包含所有的地址信息就可以了，这样做并不会导致数据冗余，例程 8-3 为 MONKEYS 表的 DDL 定义。

#### 例程 8-3 数据库 Schema

```
create table MONKEYS (
    ID bigint not null,
    NAME varchar(15),
    HOME_STREET varchar(255),
    HOME_CITY varchar(255),
    HOME_PROVINCE varchar(255),
    HOME_ZIPCODE varchar(255),
    COM_STREET varchar(255),
    COM_CITY varchar(255),
    COM_PROVINCE varchar(255),
    COM_ZIPCODE varchar(255),
    primary key (ID));
```

这样，当每次查询猴子信息时，不再需要建立表的连接。例如，以下 SQL 语句用于查询名为“Tom”的猴子的家庭地址：

```
select HOME_PROVINCE, HOME_CITY, HOME_STREET, HOME_ZIPCODE from MONKEYS
where NAME='Tom';
```

## 8.3 映射组成关系

从上面两节看出，建立对象模型和关系数据模型有着不同的出发点。对象模型是由程序代码组成的，通过细化持久化类的粒度可提高代码可重用性，简化编程。关系数据模型是由关系数据组成的。在存在数据冗余的情况下，需要把粗粒度的表

拆分成具有外键参照关系的几个细粒度的表，从而节省存储空间；另一方面，在没有数据冗余的前提下，应该尽可能减少表的数目，简化表之间的参照关系，以便提高访问数据库的速度。因此，在建立关系数据模型时，需要在节省数据存储空间和节省数据操纵时间这两者之间进行折中。

由于建立对象模型和关系数据模型的原则不一样，使得持久化类的数目往往比数据库表的数目多，而且持久化类的属性并不和表的字段一一对应。如图 8-5 所示，Monkey 类的 homeAddress 属性及 comAddress 属性均和 MONKEYS 表中的多个字段对应。

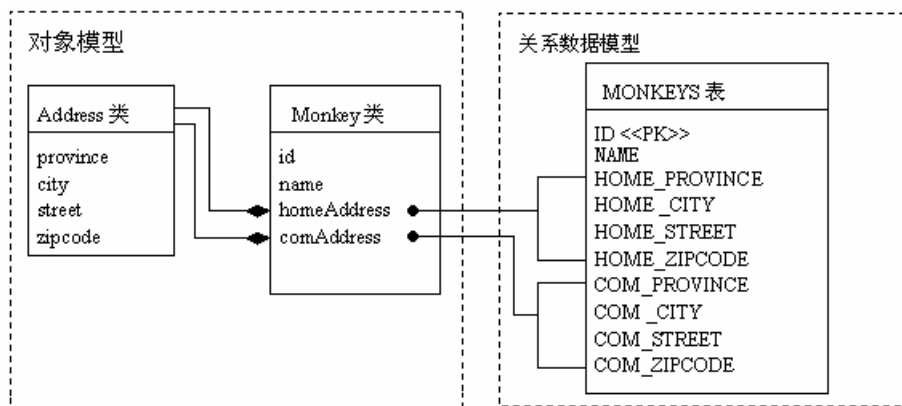


图 8-5 对象模型中类的数目比关系数据模型中表的数目多

在创建对象-关系映射文件时，不能使用<property>元素来映射 Monkey 类的 homeAddress 属性，而要使用<component>元素，映射代码如下：

```
<component name="homeAddress" class="mypack.Address">
  <parent name="monkey" />
  <property name="street" type="string" column="HOME_STREET"/>
  <property name="city" type="string" column="HOME_CITY"/>
  <property name="province" type="string" column="HOME_PROVINCE"/>
  <property name="zipcode" type="short" column="HOME_ZIPCODE"/>
</component>
```

<component>元素表明 homeAddress 属性是 Monkey 类的一个组成部分，在 Hibernate 中称为组件。<component>元素有以下两个属性。

- name: 设定被映射的持久化类的属性名，此处为 Monkey 类的 homeAddress 属性。
- class: 设定 homeAddress 属性的类型，此处表明 homeAddress 属性为 Address 类型。

<component>元素还包含一个<parent>子元素和一系列<property>子元素。<parent>元素指定 Address 类所属的整体类，这里设为 monkey，与此对应，在 Address

类中应该定义一个 `monkey` 属性，以及相关的 `getMonkey()` 和 `setMonkey()` 方法：

```
private Monkey monkey;
public Monkey getMonkey() {
    return this.monkey;
}
public void setMonkey(Monkey monkey) {
    this.monkey = monkey;
}
```

`<component>` 元素的 `<property>` 子元素用来配置组件类的属性和表中字段的映射。例如，`homeAddress` 组件的 `city` 属性和 `MONKEYS` 表的 `HOME_CITY` 字段映射，而 `comAddress` 组件的 `city` 属性和 `MONKEYS` 表的 `COM_CITY` 字段映射。例程 8-4 是 `Monkey.hbm.xml` 文件的源代码。

例程 8-4 Monkey.hbm.xml

```
<hibernate-mapping >

  <class name="mypack.Monkey" table="MONKEYS" >
    <id name="id" type="long" column="ID">
      <generator class="increment"/>
    </id>

    <property name="name" type="string" >
      <column name="NAME" length="15" />
    </property>

    <component name="homeAddress" class="mypack.Address">
      <parent name="monkey" />
      <property name="street" type="string" column="HOME_STREET"/>
      <property name="city" type="string" column="HOME_CITY"/>
      <property name="province" type="string" column="HOME_PROVINCE"/>
      <property name="zipcode" type="string" column="HOME_ZIPCODE"/>
    </component>

    <component name="comAddress" class="mypack.Address">
      <parent name="monkey" />
      <property name="street" type="string" column="COM_STREET"/>
      <property name="city" type="string" column="COM_CITY"/>
      <property name="province" type="string" column="COM_PROVINCE"/>
      <property name="zipcode" type="string" column="COM_ZIPCODE"/>
    </component>
  </class>
</hibernate-mapping>
```



```

    </component>
  </class>
</hibernate-mapping>

```

### 8.3.1 区分值（Value）类型和实体（Entity）类型

从本章 8.1 节的例程 8-2 看出，Address 类没有 OID，这是 Hibernate 组件的一个重要特征。由于 Address 类没有 OID，因此不能通过 Session 来单独保存、更新、删除或加载一个 Address 对象，例如，以下每行代码都会抛出 org.hibernate.MappingException 异常：

```

session.load(Address.class,new Long(1));
session.save(address);
session.update(address);
session.delete(address);

```

Hibernate 执行以上代码时抛出 MappingException 异常，错误原因为“Unknown entity class: mypack.Address”。为何称 Address 类是未知的实体类呢？这是因为 Hibernate 把持久化类的属性分为两种：值（Value）类型和实体（Entity）类型。值类型和实体类型的最重要的区别是前者没有 OID，不能被单独持久化，它的生命周期依赖于所属的持久化类的对象的生命周期，组件类型就是一种值类型；而实体类型有 OID，可以被单独持久化。假定 Monkey 类有以下属性：

```

private String name;
private int age;
private Date birthday;

// Monkey 和 Address 类是组成关系
private Address homeAddress;
private Address comAddress;

//Monkey 和 Team 类是多对一关联关系
private Team team;

//Monkey 和 Teacher 类是一对多关联关系
private Set teachers;

```

在以上属性中，name、age、birthday、homeAddress 及 comAddress 都是值类型属性，而 team 是实体类型属性，teachers 集合中的 Teacher 对象也是实体类型属性。当删除一个 Monkey 持久化对象时，Hibernate 会从数据库中删除所有值类型属性对应的数据，但是实体类型属性对应的数据有可能依然保留在数据库中，也有可能被删除，这取决于是否在映射文件中设置了级联删除。假如对 teachers 集合设置了级联删除，那么删除 Monkey 对象时，也会删除 teachers 集合中的所有 Teacher 对象。

假如没有对 `team` 属性设置级联删除，那么删除一个 `Monkey` 对象时，`team` 属性引用的 `Team` 对象依然存在。

`Address` 类作为值类型没有 `OID`，因此不能建立从其他持久化类到 `Address` 类的关联关系（注意关联是有方向性的）。假如在 `Monkey.hbm.xml` 文件中按如下方式映射 `team` 属性和 `homeAddress` 属性：

```
<many-to-one
    name="team"
    column="TEAM_ID"
    class="mypack.Team"
/>

<many-to-one
    name="homeAddress"
    column="ADDRESS_ID"
    class="mypack.Address"
/>
```

以上代码对 `team` 属性做了正确的映射，但是对 `homeAddress` 属性的映射是不正确的。以上代码意味着 `MONKEYS` 表有一个外键 `ADDRESS_ID` 参照 `Address` 类的对应表的主键，而实际上 `Address` 类在数据库中根本没有对应的表。

另一方面，可以建立组件类到其他持久化类的关联，在本章 8.4 节的例子中，`CpuBox` 是一个组件类，它还和 `Vendor` 类关联。

此外，`Address` 类作为值类型没有单独的对象-关系映射文件，即无须为 `Address` 类创建 `Address.hbm.xml` 文件。在 `Monkey.hbm.xml` 文件的 `<component>` 元素中包含了 `Address` 类的映射信息。

### 8.3.2 在应用程序中访问具有组成关系的持久化类

本章范例程序位于配套光盘的 `sourcecode\chapter8` 目录下。在 DOS 下转到本例的根目录 `chapter8`，输入命令：`ant run`，该命令将运行 `BusinessService` 类，它的源程序参见例程 8-5。

例程 8-5 `BusinessService` 类

```
package mypack;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class BusinessService{
    public static SessionFactory sessionFactory;
    static{.....} /* 初始化 Hibernate，创建 SessionFactory 实例 */
```

```

/** 保存一个 Monkey 对象 */
public void saveMonkey(){.....}

/** 单独保存一个 Address 对象 */
public void saveAddressSeparately(){.....}

/** 保存一个 Address 为 null 的 Monkey 对象 */
public void saveMonkeyWithNoAddress(){.....}

/** 按照 Monkey ID 查询 Monkey 对象 */
public Monkey findMonkey(long monkey_id){.....}

/** 打印 Monkey 的地址信息 */
public void printMonkeyAddress(Monkey monkey){.....}

/** 删除一个 Monkey 对象*/
public void deleteMonkey(Monkey monkey){.....}

public void test(){
    saveMonkey();
    saveAddressSeparately();
    saveMonkeyWithNoAddress();
    Monkey monkey=findMonkey(1);
    printMonkeyAddress(monkey);
    monkey=findMonkey(2);
    printMonkeyAddress(monkey);
    deleteMonkey(monkey);
}

public static void main(String args[]){
    new BusinessService().test();
    sessionFactory.close();
}
}

```

BusinessService 类的 main()方法调用 test()方法，test()方法又依次调用以下方法。

(1)saveMonkey():先创建一个 Monkey 对象和两个 Address 对象，建立 Monkey 和 Address 对象之间的组成关系，然后保存 Monkey 对象：

```

tx = session.beginTransaction();
Monkey monkey=new Monkey();
Address homeAddress=new Address("provincel","city1","street1",
    "100001",monkey);
Address comAddress=new Address("province2","city2","street2",
    "200002",monkey);

```

```
monkey.setName("Tom");
monkey.setHomeAddress(homeAddress);
monkey.setComAddress(comAddress);

session.save(monkey);
tx.commit();
```

当 Hibernate 持久化 Monkey 对象时，会自动保存两个 Address 组件类对象。  
Hibernate 执行以下 SQL 语句：

```
insert into MONKEYS
(ID,NAME,HOME_PROVINCE,HOME_CITY,HOME_STREET,HOME_ZIPCODE,
COM_PROVINCE,COM_CITY,COM_STREET,COM_ZIPCODE)
values(1, 'Tom','province1','city1','street1','100001','province2',
'city2','street2','200002');
```

(2) saveAddressSeparately(): 这个方法试图单独保存一个 Address 对象：

```
tx = session.beginTransaction();
Address address=new Address("province1","city1","street1","100001",
null);
session.save(address);
tx.commit();
```

Hibernate 不允许单独持久化组件类对象，执行以上代码会抛出以下异常：

```
[java] org.hibernate.MappingException: Unknown entity class: mypack.Address
```

(3) saveMonkeyWithNoAddress(): 该方法先创建一个 Monkey 对象，它的 homeAddress 属性引用一个 Address 对象，但是该 Address 对象的所有属性都是 null，Monkey 对象的 comAddress 属性为 null。最后保存这个 Monkey 对象。

```
tx = session.beginTransaction();

Monkey monkey=new Monkey("Mike",new Address(null,null,null,null,
null),null);
session.save(monkey);
tx.commit();
```

当 Hibernate 持久化 Monkey 对象时，执行以下 SQL 语句：

```
insert into MONKEYS
(ID,NAME,HOME_PROVINCE,HOME_CITY,HOME_STREET,HOME_ZIPCODE,
COM_PROVINCE,COM_CITY,COM_STREET,COM_ZIPCODE)
values(2, 'Mike','null','null','null','null','null','null',
'null','null');
```

(4) findMonkey(): 按照参数指定的 OID 来查询 Monkey 对象

(5) printMonkeyAddress(): 打印参数指定的 Monkey 对象的所有地址信息。

由于 Address 对象是 Monkey 对象的组件，因此只要调用 monkey.getHomeAddress() 方法，就可以方便地从 Monkey 对象导航到 Address 对象：

```
Address homeAddress=monkey.getHomeAddress();
Address comAddress=monkey.getComAddress();

if(homeAddress==null)
    System.out.println("Home Address of "+monkey.getName()+" is null.");
else
    System.out.println("Home Address of "+monkey.getName()+" is: "
        +homeAddress.getProvince()+" "
        +homeAddress.getCity()+" "
        +homeAddress.getStreet());

if(comAddress==null)
    System.out.println("Company Address of "+monkey.getName()+" is null.");
else
    System.out.println("Company Address of "+monkey.getName()+" is: "
        +comAddress.getProvince()+" "
        +comAddress.getCity()+" "
        +comAddress.getStreet());
```

当 printMonkeyAddress ()打印 OID 为 1 的 Monkey 对象时，输出的信息为：

```
Home Address of Tom is: provincel city1 street1
Company Address of Tom is: province2 city2 street2
```

当 printMonkeyAddress ()打印 OID 为 2 的 Monkey 对象时，输出的信息为：

```
Home Address of Mike is null.
Company Address of Mike is null.
```

OID 为 2 的 Monkey 对象是在 saveMonkeyWithNoAddress()方法中创建的，在保存 Monkey 对象时，它的 homeAddress 属性引用一个 Address 实例，但是这个 Address 实例的所有属性都是 null。当从数据库中加载这个 Monkey 对象时，由于数据库中 HOME\_PROVINCE、HOME\_CITY、HOME\_STREET 和 HOME\_ZIPCODE 字段都是 null，因此 Hibernate 把 homeAddress 属性赋值为 null。

(6) deleteMonkey(): 删除参数指定的 Monkey 对象：

```
tx = session.beginTransaction();
session.delete(monkey);
tx.commit();
```

当 Hibernate 删除 Monkey 对象时，会自动删除它包含的 Address 类组件对象。Hibernate 执行的 SQL 语句为：

```
delete from MONKEYS where ID=2;
```

## 8.4 映射复合组成关系

一个 Hibernate 组件可以包含其他 Hibernate 组件，或者和其他实体类关联。例如在图 8-6 中，CpuBox 类是 Computer 类的一个组件，而 GraphicsCard 类是 CpuBox 类的组件，此外，CpuBox 类还和 Vendor（供应商）类多对一关联。

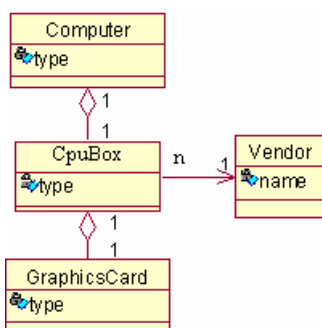


图 8-6 Computer 类和它的组件类

图 8-6 中有 4 个类：Computer 类、CpuBox 类、GraphicsCard 类和 Vendor 类。这些类的源文件参见配套光盘。值得注意的是，在图 8-6 中，只有 Computer 类和 Vendor 类是实体类，而 CpuBox 类和 GraphicsCard 类都是值类型的组件类。因此，只需为 Computer 类和 Vendor 类提供对象-关系映射文件。例程 8-6 列出了 Computer.hbm.xml 文件的源代码，Vendor.hbm.xml 文件的源代码很简单，可以参考本书配套光盘。

### 例程 8-6 Computer.hbm.xml 文件

```

<hibernate-mapping>

  <class name="mypack.Computer" table="COMPUTERS">
    <id name="id" type="long" column="ID">
      <generator class="increment"/>
    </id>

    <property name="type" type="string">
      <column name="COMPUTER_TYPE"/>
    </property>

    <component name="cpuBox" class="mypack.CpuBox">
      <parent name="computer"/>

      <property name="type" type="string">
        <column name="CPUBOX_TYPE"/>
      </property>
    </component>
  </class>

```

```

<component name="graphicsCard" class="mypack.GraphicsCard">
  <parent name="cpuBox" />

  <property name="type" type="string" >
    <column name="GRAPHICSCARD_TYPE"/>
  </property>

</component>

<many-to-one
  name="vendor"
  column="CPUBOX_VENDOR_ID"
  class="mypack.Vendor"
  not-null="true"
/>
</component>
</class>
</hibernate-mapping>

```

在例程 8-6 中，<component>元素中嵌套了<component>元素和<many-to-one>元素。以下例程 8-7 为 VENDORS 表和 COMPUTERS 表的 DDL 定义。

**例程 8-7 数据库 Schema**

```

create table COMPUTERS (ID bigint not null, COMPUTER_TYPE varchar(15),
CPUBOX_TYPE varchar(15), GRAPHICSCARD_TYPE varchar(15),
CPUBOX_VENDOR_ID bigint not null, primary key (ID));

create table VENDORS (ID bigint not null, TYPE varchar(15), primary key (ID));

alter table COMPUTERS add index IDX_VENDOR (CPUBOX_VENDOR_ID),
add constraint FK_VENDOR foreign key (CPUBOX_VENDOR_ID)
references VENDORS (ID);

```

从例程 8-7 看出，COMPUTERS 表的 CPUBOX\_VENDOR\_ID 外键参照 VENDORS 表的 ID 主键。

## 8.5 小结

本章主要以 Monkey 和 Address 类为例介绍了组成关系的映射，Hibernate 用 <component>元素来映射 Monkey 类的 homeAddress 和 comAddress 属性。Address 类作为 Hibernate 的组件，具有以下特征：

- 没有 OID，在数据库中没有对应的表。
- 不需要单独创建 Address 类的映射文件。

- 不能单独持久化 Address 对象。Address 对象的生命周期依赖于 Monkey 对象的生命周期。
- 其他持久化类不允许关联 Address 类。
- Address 类可以关联其他持久化类。

在<component>元素中还能嵌套<component>元素，它用于映射复合组成关系。



## 第9章 Hibernate的映射类型

Monkey 类有一个 String 类型的 name 属性,与此对应,在 MONKEYS 表中有一个 VARCHAR 类型的 NAME 字段。在 Monkey.hbm.xml 对象-关系映射文件中,以下代码把 Monkey 类的 name 属性映射到 MONKEYS 表的 NAME 字段:

```
<property name="name" type="string" column="NAME" />
```

以上<property>元素的 type 属性的值为“string”,它既不是 Java 类型,也不是 SQL 类型,而是 Hibernate 映射类型。Hibernate 映射类型是 Java 类型和 SQL 类型的桥梁。

对于以上映射代码,由于<property>元素的 type 属性的值为“string”,Hibernate 就能自动推断出 Monkey 类的 name 属性为 String 类型,并且 MONKEYS 表的 NAME 字段为 VARCHAR 类型。

Hibernate 映射类型分为两种:内置映射类型和客户化映射类型。内置映射类型负责把一些常见的 Java 类型映射到相应的 SQL 类型;此外,Hibernate 还允许用户实现 UserType 接口,来灵活地定制客户化映射类型。客户化映射类型能够把用户定义的 Java 类型映射到数据库表的相应字段。

### 9.1 Hibernate的内置映射类型

Hibernate 的内置映射类型通常使用和 Java 类型相同的名字,它能够把 Java 基本类型、Java 时间和日期类型、Java 大对象类型及 JDK 中常用 Java 类型映射到相应的标准 SQL 类型。

#### 9.1.1 Java基本类型的Hibernate映射类型

表 9-1 列出了 Hibernate 映射类型、对应的 Java 基本类型(或者它们的包装类),以及对应的标准 SQL 类型。

表 9-1 Hibernate 映射类型、对应的 Java 基本类型及对应的标准 SQL 类型

Hibernate 映射类型	Java 类型	标准 SQL 类型	大小和取值范围
integer 或者 int	int 或者 java.lang.Integer	INTEGER	4 字节, $-2^{31} \sim 2^{31} - 1$
long	long 或者 java.lang.Long	BIGINT	8 字节, $-2^{63} \sim 2^{63} - 1$
short	short 或者 java.lang.Short	SMALLINT	2 字节, $-2^{15} \sim 2^{15} - 1$
byte	byte 或者 java.lang.Byte	TINYINT	1 字节, $-128 \sim 127$

(续表)

Hibernate 映射类型	Java 类型	标准 SQL 类型	大小和取值范围
float	float 或者 java.lang.Float	FLOAT	4 字节, 单精度浮点数
double	double 或者 java.lang.Double	DOUBLE	8 字节, 双精度浮点数
big_decimal	java.math.BigDecimal	NUMERIC	NUMERIC(8,2), 表示共 8 位数字, 其中小数部分占 2 位
character	char 或者 java.lang.Character, java.lang.String	CHAR(1)	定长字符
string	java.lang.String	VARCHAR	变长字符串
boolean	boolean 或者 java.lang.Boolean	BIT	布尔类型
yes_no	boolean 或者 java.lang.Boolean	CHAR(1)('Y' 或者 'N')	布尔类型
true_false	boolean 或者 java.lang.Boolean	CHAR(1)('T' 或者 'F')	布尔类型

### 9.1.2 Java时间和日期类型的Hibernate映射类型

在 Java 中, 代表时间和日期的类型包括: java.util.Date 和 java.util.Calendar。此外, 在 JDBC API 中还提供了 3 个扩展了 java.util.Date 类的子类: java.sql.Date、java.sql.Time 和 java.sql.Timestamp, 这 3 个类分别和标准 SQL 类型中的 DATE、TIME 和 TIMESTAMP 类型对应。

表 9-2 列出了 Hibernate 映射类型、对应的 Java 时间和日期类型, 以及对应的标准 SQL 类型。

表 9-2 Hibernate 映射类型、对应的 Java 时间和日期类型及对应的标准 SQL 类型			
映射类型	Java 类型	标准 SQL 类型	描述
date	java.util.Date 或者 java.sql.Date	DATE	代表日期, 形式为: YYYY-MM-DD
time	java.util.Date 或者 java.sql.Time	TIME	代表时间, 形式为: HH:MM:SS
timestamp	java.util.Date 或者 java.sql.Timestamp	TIMESTAMP	代表时间和日期, 形式为: YYYYMMDDHHMMSS
calendar	java.util.Calendar	TIMESTAMP	同上
calendar_date	java.util.Calendar	DATE	代表日期, 形式为: YYYY-MM-DD

### 9.1.3 Java大对象类型的Hibernate映射类型

在 Java 中, java.lang.String 可用于表示长字符串 (长度超过 255), 字节数组 byte[] 可用于存放图片或长文件的二进制数据。此外, 在 JDBC API 中还提供了 java.sql.Clob 和 java.sql.Blob 类型, 它们分别和标准 SQL 中的 CLOB 和 BLOB 类型对应。CLOB 表示字符串大对象 (Character Large Object), BLOB 表示二进制大对象 (Binary Large Object)。表 9-3 列出了 Hibernate 映射类型、对应的 Java 大对象

类型，以及对应的标准 SQL 类型。

表 9-3 Hibernate 映射类型、对应的 Java 大对象类型及对应的标准 SQL 类型			
映 射 类 型	Java 类型	标准 SQL 类型	描 述
binary	byte[]	VARBINARY（或者 BLOB）	二进制数据
text	java.lang.String	CLOB	长字符串
serializable	实现 java.io.Serializable 接口的任意一个 Java 类	VARBINARY（或者 BLOB）	二进制序列化数据
clob	java.sql.Clob	CLOB	长字符串
blob	Java.sql.Blob	BLOB	二进制数据

9.1.4 JDK自带的个别Java类的Hibernate映射类型

表 9-4 列出了用于映射 JDK 自带的个别 Java 类的 Hibernate 映射类型，与此对应的标准 SQL 类型均为 VARCHAR 类型。

表 9-4 Hibernate 映射类型、对应的 Java 类型及对应的标准 SQL 类型		
映 射 类 型	Java 类型	标准 SQL 类型
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

9.1.5 使用Hibernate内置映射类型

Hibernate 的内置映射类型、Java 类型和标准 SQL 类型三者之间的关系是固定的。如图 9-1 所示，Monkey 类的 id 属性为 java.lang.Long，而 MONKEYS 表的 ID 字段为 BIGINT 类型，那么应该用 Hibernate 的 long 类型来映射它们；Monkey 类的 name 属性为 java.lang.String，而 MONKEYS 表的 NAME 字段为 VARCHAR 类型，那么应该用 Hibernate 的 string 类型来映射。

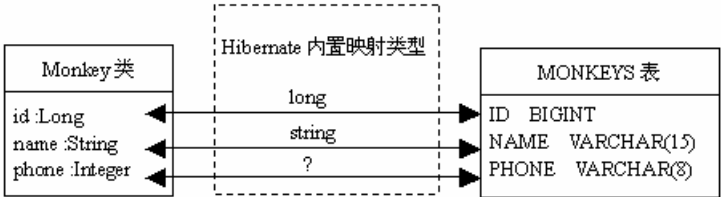


图 9-1 Monkey 类和 MONKEYS 表的映射

由于 Hibernate 的内置映射类型、Java 类型和标准 SQL 类型三者之间的关系是固定的，因此在映射持久化类的属性时，有时可以省略设置 Hibernate 映射类型，例如以下两种方式是等价的：

```
<property name="name" type="string" column="NAME" />
```

或者:

```
<property name="name" column="NAME" />
```

如果没有指定映射类型, Hibernate 会运用反射机制, 先判别 Monkey 类的 name 属性的 Java 类型, 然后采用与此 Java 类型对应的默认 Hibernate 映射类型。例如与 java.lang.String 类型对应的默认 Hibernate 映射类型为 “string”。

但是, 在一个 Java 类型对应多个 Hibernate 映射类型的场合, 有时必须显式指定 Hibernate 映射类型。例如, 如果持久化类的一个属性为 java.util.Date 类型, 对应的 Hibernate 映射类型可以是 date、time 或 timestamp。此时必须根据对应的数据库表的字段的 SQL 类型, 来确定 Hibernate 映射类型。如果字段为 DATE 类型, 那么 Hibernate 映射类型为 date; 如果字段为 TIME 类型, 那么 Hibernate 映射类型为 time; 如果字段为 TIMESTAMP 类型, 那么 Hibernate 映射类型为 timestamp。

在图 9-1 中, Monkey 类的 phone 属性为 java.lang.Integer, 而对应的表的 PHONE 字段为 VARCHAR 类型, 那么应该用 Hibernate 的什么类型来映射呢? 下面的两种映射方式都是不正确的:

```
<property name="phone" type="integer" column="PHONE" />
```

或者:

```
<property name="phone" type="string" column="PHONE" />
```

当 Hibernate 持久化 Monkey 对象, 无法自动把 java.lang.Integer 类型的 phone 属性转换为字符串类型, 因此会抛出 ClassCastException。在下一节, 将介绍如何利用 Hibernate 的客户化映射类型, 把持久化类的任意类型的属性映射到数据库中。

## 9.2 客户化映射类型

Hibernate 提供了客户化映射类型接口, 允许用户以编程的方式创建自定义的映射类型, 以便把持久化类的任意类型的属性映射到数据库中。例程 9-1 的 PhoneUserType 实现了 org.hibernate.usertype.UserType 接口, 它能够把 Monkey 类的 Integer 类型的 phone 属性映射到 MONKEYS 表的 VARCHAR 类型的 PHONE 字段。

例程 9-1 PhoneUserType

```
package mypack;

import org.hibernate.*;
import org.hibernate.usertype.*;
import java.sql.*;
```

```
import java.io.Serializable;

public class PhoneUserType implements UserType {

    private static final int[] SQL_TYPES = {Types.VARCHAR};

    public int[] sqlTypes() { return SQL_TYPES; }//(1)

    public Class returnedClass() { return Integer.class; }//(2)

    public boolean isMutable() { return false; }//(3)

    public Object deepCopy(Object value) { //(4)
        return value;
    }

    public boolean equals(Object x, Object y) { //(5)
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }

    public int hashCode(Object x){ //(6)
        return x.hashCode();
    }

    public Object nullSafeGet(ResultSet resultSet, String[] names,
        Object owner)
        throws HibernateException, SQLException { //(7)

        if (resultSet.isNull()) return null;
        String phone = resultSet.getString(names[0]);
        return new Integer(phone);
    }

    public void nullSafeSet(PreparedStatement statement, Object value,
        int index)
        throws HibernateException, SQLException { //(8)
        if (value == null) {
            statement.setNull(index, Types.VARCHAR);
        } else {
            String phone=((Integer)value).toString();
            statement.setString(index, phone);
        }
    }
}
```

```

    public Object assemble(Serializable cached, Object owner){ //(9)
        return cached;
    }

    public Serializable disassemble(Object value) { //(10)
        return (Serializable)value;
    }

    public Object replace(Object original, Object target, Object owner){
        //(11)
        return original;
    }
}

```

PhoneUserType 实现了 org.hibernate.UserType 接口中的所有方法，下面解释这些方法的作用。

#### (1) sqlTypes()方法

设置 MONKEYS 表的 PHONE 字段的 SQL 类型，它是 VARCHAR 类型：

```

private static final int[] SQL_TYPES = {Types.VARCHAR};
public int[] sqlTypes() { return SQL_TYPES; }

```

#### (2) returnedClass()方法

设置 Monkey 类的 phone 属性的 Java 类型，它是 Integer 类型：

```

public Class returnedClass() { return Integer.class; }

```

#### (3) isMutable()方法

Hibernate 调用 isMutable()方法来了解 Integer 类是否为可变类。本例的 Integer 类是不可变类，因此返回 false。Hibernate 处理不可变类型的属性时，会采取一些性能优化措施。本章 9.5 节归纳了可变类与不可变类的区别。

```

public boolean isMutable() { return false; }

```

#### (4) deepCopy(Object value)方法

Hibernate 调用 deepCopy(Object value)方法来生成 phone 属性的快照（也称为深度复制）。deepCopy()方法的 value 参数代表 Integer 类型的 phone 属性。由于 Integer 类是不可变类，因此本方法直接返回 value 参数：

```

public Object deepCopy(Object value) {
    return value;
}

```

对于可变类，以上 deepCopy()方法必须返回参数的复制值，参见本章 9.5 节。

#### (5) equals(Object x, Object y)方法

Hibernate 调用 `equals(Object x, Object y)` 方法来比较 Monkey 类的 `phone` 属性的当前值是否和它的快照相同。该方法的参数 `x` 代表 `phone` 属性的当前值，参数 `y` 代表由 `deepCopy()` 方法生成的 `phone` 属性的快照：

```
public boolean equals(Object x, Object y) {
    if (x == y) return true;
    if (x == null || y == null) return false;
    return x.equals(y);
}
```

#### (6) `hashCode(Object x)` 方法

Hibernate 调用 `hashCode(Object x)` 方法来获得 Monkey 类的 `phone` 属性的哈希码。该方法的参数 `x` 代表 `phone` 属性的当前值：

```
public int hashCode(Object x){
    return x.hashCode();
}
```

#### (7) `nullSafeGet(ResultSet resultSet, String[] names, Object owner)` 方法

当 Hibernate 从数据库加载 Monkey 对象时，调用 `nullSafeGet()` 方法来取得 `phone` 属性值。参数 `resultSet` 为 JDBC 查询结果集，参数 `names` 为存放了表字段名的数组，此处为 `{"PHONE"}`，参数 `owner` 代表 `phone` 属性的宿主，此处为 Monkey 对象。

```
public Object nullSafeGet(ResultSet resultSet, String[] names,
    Object owner)
    throws HibernateException, SQLException {

    if (resultSet.isNull()) return null;
    String phone = resultSet.getString(names[0]);
    return new Integer(phone);
}
```

在 `nullSafeGet()` 方法中，先从 `ResultSet` 中读取 `PHONE` 字段的值，然后把它转换为 `Integer` 对象，最后将它作为 `phone` 属性值返回。

#### (8) `nullSafeSet(PreparedStatement statement, Object value, int index)` 方法

当 Hibernate 把 Monkey 对象持久化到数据库中时，调用 `nullSafeSet()` 方法来把 `phone` 属性添加到 SQL insert 语句中。`statement` 参数包含了预定义的 SQL insert 语句，参数 `value` 代表 `phone` 属性，参数 `index` 代表把 `phone` 属性插入到 SQL insert 语句中的位置：

```
public void nullSafeSet(PreparedStatement statement, Object value, int
index)
    throws HibernateException, SQLException {
    if (value == null) {
```

```

        statement.setNull(index, Types.VARCHAR);
    } else {
        String phone=((Integer)value).toString();
        statement.setString(index, phone);
    }
}

```

在 `nullSafeSet()` 方法中, 参数 `value` 代表 `phone` 属性。因此, 先把 `Integer` 类型的 `value` 转换为 `String` 类型, 然后把它添加到 `JDBC Statement` 中。

#### (9) `assemble(Serializable cached, Object owner)` 方法

当 Hibernate 把第二级缓存中的 `Monkey` 对象加载到 `Session` 缓存中时, 调用 `assemble()` 方法来获得 `phone` 属性的反序列化数据。Hibernate 的第二级缓存中存放了 `Monkey` 对象的序列化数据。参数 `cached` 代表 `phone` 属性的反序列化数据, 参数 `owner` 代表 `phone` 属性的宿主, 此处为 `Monkey` 对象。如果参数 `cached` 为不可变类型, 可以直接返回 `cached` 参数; 如果参数 `cached` 为可变类型, 则应该返回参数 `cached` 的快照 (即调用 `deepCopy(cached)` 方法的返回值)。

```

public Object assemble(Serializable cached, Object owner){
    return cached;
}

```

#### (10) `disassemble(Object value)` 方法

当 Hibernate 把 `Session` 缓存中的 `Monkey` 对象保存到第二级缓存中时, 调用 `disassemble()` 方法来获得 `phone` 属性的序列化数据。参数 `value` 代表 `phone` 属性的序列化数据。如果参数 `value` 为不可变类型, 可以直接返回 `value` 参数; 如果参数 `value` 为可变类型, 则应该返回参数 `value` 的快照 (即调用 `deepCopy(value)` 方法的返回值)。

```

public Serializable disassemble(Object value) {
    return (Serializable)value;
}

```

#### (11) `replace(Object original, Object target, Object owner)` 方法

当 `Session` 的 `merge()` 方法把一个 `MonkeyA` 游离对象融合到 `MonkeyB` 持久化对象中时, 会调用此 `replace()` 方法来获得用于替代 `MonkeyB` 对象的 `phone` 属性的值。参数 `original` 代表 `MonkeyA` 游离对象的 `phone` 属性, 参数 `target` 代表 `MonkeyB` 对象的 `phone` 属性, 参数 `owner` 代表 `MonkeyA` 对象。如果参数 `original` 为不可变类型, 可以直接返回 `original` 参数; 如果参数 `original` 为可变类型, 则应该返回参数 `original` 的快照 (即调用 `deepCopy(original)` 方法的返回值)。

```

public Object replace(Object original, Object target, Object owner){
    return original;
}

```



创建了以上的 PhoneUserType 类后,就可以按以下方式映射 Monkey 类的 phone 属性:

```
<property name="phone" type="mypack.PhoneUserType" column="PHONE" />
```

PhoneUserType 不仅可以用来映射 phone 属性,事实上,它能够把持久化类的任何一个 Integer 类型的属性映射到数据库表中 VARCHAR 类型的字段。

## 9.3 用客户化映射类型取代Hibernate组件

本书第 8 章介绍了用 Hibernate 组件来映射 Monkey 类的 Address 类型的 homeAddress 属性和 comAddress 属性。本节自定义了一个 AddressUserType 映射类型,它也能把 Address 类型的属性映射到数据库。

第 8 章的 Address 类为可变类。为了演示不可变类的特性,本章特意把 Address 类设计为不可变类。所谓不可变类,是指当创建了这种类的实例后,就不允许修改它的属性。在 Java API 中,所有的基本类型的包装类,如 Integer 和 Long 类,都是不可变类,java.lang.String 也是不可变类。在创建用户自己的不可变类时,可以考虑采用以下的设计模式:

- 把属性定义为 private final 类型。
- 不对外公开用于修改属性的 setXXX()方法。
- 只对外公开用于读取属性的 getXXX()方法。
- 允许在构造方法中设置所有属性。
- 覆盖 Object 类的 equals()和 hashCode()方法,在 equals()方法中根据对象的属性值来比较两个对象是否相等,保证用 equals()方法比较相等的两个对象的 hashCode()方法的返回值也相等。

以下例程 9-2 是 Address 类的源程序。

例程 9-2 Address.java

```
package mypack;

public class Address{

    private final String province;
    private final String city;
    private final String street;
    private final String zipcode;

    public Address(String province, String city, String street,
        String zipcode){
```

```
        this.street = street;
        this.city = city;
        this.province = province;
        this.zipcode = zipcode;
    }

    public String getProvince() {
        return this.province;
    }

    public String getCity() {
        return this.city;
    }

    public String getStreet() {
        return this.street;
    }

    public String getZipcode() {
        return this.zipcode;
    }

    public boolean equals(Object o){
        if (this == o) return true;
        if (!(o instanceof Address)) return false;

        final Address address = (Address) o;

        if(!province.equals(address.province)) return false;
        if(!city.equals(address.city)) return false;
        if(!street.equals(address.street)) return false;
        if(!zipcode.equals(address.zipcode)) return false;
        return true;
    }

    public int hashCode(){
        int result;
        result= (province==null?0:province.hashCode());
        result = 29 * result + (city==null?0:city.hashCode());
        result = 29 * result + (street==null?0:street.hashCode());
        result = 29 * result + (zipcode==null?0:zipcode.hashCode());
        return result;
    }
}
```

由于 Address 类是不可变类，因此创建了 Address 类的实例后，就无法修改它

的属性。如果要修改 **Monkey** 对象的 **comAddress** 属性，必须使它引用一个新的 **Address** 实例：

```
Address comAddress=new Address("comProvince","comCity","comStreet","200002");
monkey.setComAddress(comAddress); //最初的 Address 实例

//创建一个新的 Address 实例
comAddress=new Address("comProvinceNew","comCityNew","comStreetNew","200002");
//修改 Monkey 对象的 comAddress 属性
monkey.setComAddress(comAddress);
```

例程 9-3 是 **AddressUserType** 类的源程序，它实现了 **UserType** 接口。

**例程 9-3** AddressUserType.java

```
package mypack;
import org.hibernate.*;
import org.hibernate.usertype.*;
import java.sql.*;
import java.io.Serializable;

public class AddressUserType implements UserType {

    /** 设置和 Address 类的 4 个属性 province、city、street 和 zipcode 对应的字段的 SQL 类型，它们均为 VARCHAR 类型 */
    private static final int[] SQL_TYPES =
    {Types.VARCHAR,Types.VARCHAR,Types.VARCHAR,Types.VARCHAR};

    public int[] sqlTypes() { return SQL_TYPES; }

    /** 设置 AddressUserType 所映射的 Java 类: Address 类 */
    public Class returnedClass() { return Address.class; }

    /** 指明 Address 类是不可变类 */
    public boolean isMutable() { return false; }

    /** 返回 Address 对象的快照，由于 Address 类是不可变类，因此直接将参数代表的 Address 对象返回 */
    public Object deepCopy(Object value) {
        return value;
    }

    /** 比较一个 Address 对象是否和它的快照相同 */
    public boolean equals(Object x, Object y) {
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }
}
```

```

public int hashCode(Object x){
    return x.hashCode();
}

/** 从 JDBC ResultSet 中读取 province、city、street 和 zipcode,
    然后构造一个 Address 对象*/
public Object nullSafeGet(ResultSet resultSet,String[] names,
    Object owner)
    throws HibernateException, SQLException {
    if (resultSet.isNull()) return null;
    String province = resultSet.getString(names[0]);
    String city = resultSet.getString(names[1]);
    String street = resultSet.getString(names[2]);
    String zipcode = resultSet.getString(names[3]);
    return new Address(province,city,street,zipcode);
}

/** 把 Address 对象的属性添加到 JDBC PreparedStatement 中 */
public void nullSafeSet(PreparedStatement statement,Object value,
    int index)
    throws HibernateException, SQLException {
    if (value == null) {
        statement.setNull(index, Types.VARCHAR);
        statement.setNull(index+1, Types.VARCHAR);
        statement.setNull(index+2, Types.VARCHAR);
        statement.setNull(index+3, Types.VARCHAR);
    } else {
        Address address=(Address)value;
        statement.setString(index, address.getProvince());
        statement.setString(index+1, address.getCity());
        statement.setString(index+2, address.getStreet());
        statement.setString(index+3, address.getZipcode());
    }
}

public Object assemble(Serializable cached, Object owner){
    return cached;
}

public Serializable disassemble(Object value) {
    return (Serializable)value;
}

public Object replace(Object original,Object target,Object owner){
    return original;
}
}

```

创建了以上的 AddressUserType 后，就可以按以下方式映射 Monkey 类的 homeAddress 和 comAddress 属性：

```
<property name="homeAddress" type="mypack.AddressUserType" >
    <column name="HOME_PROVINCE" />
    <column name="HOME_CITY" />
    <column name="HOME_STREET" />
    <column name="HOME_ZIPCODE" />
</property>

<property name="comAddress" type="mypack.AddressUserType" >
    <column name="COM_PROVINCE" />
    <column name="COM_CITY" />
    <column name="COM_STREET" />
    <column name="COM_ZIPCODE" />
</property>
```

Hibernate 组件和客户化映射类型都是值类型，在某些情况下能够完成同样的功能，到底选择何种方式，取决于用户自己的喜好。总之，Hibernate 组件采用的是 XML 配置方式，因此具有较好的可维护性。客户化映射类型采用的是编程方式，能够完成更加复杂灵活的映射。

9.4 运行范例程序

本章共创建了两个客户化映射类型：AddressUserType 和 PhoneUserType，图 9-2 显示了这两个映射类型的作用。

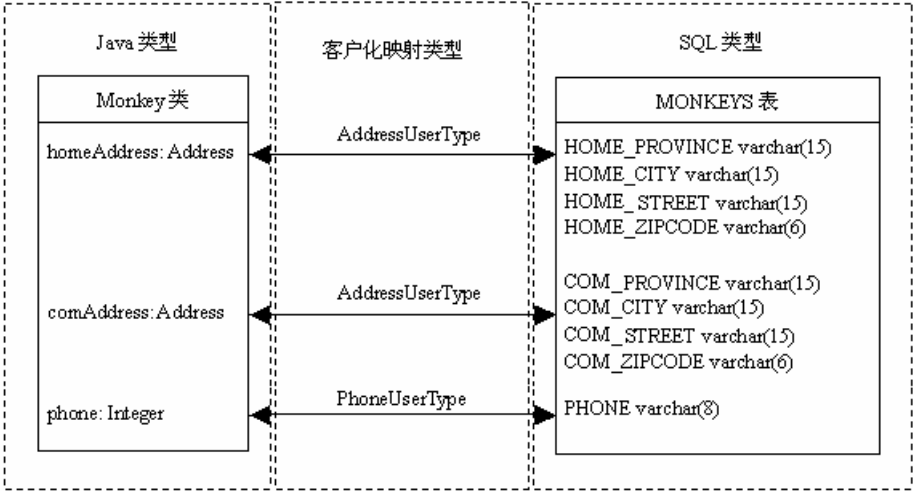


图 9-2 范例中 Java 类型、客户化映射类型和 SQL 类型的对应关系

例程 9-4 是 Monkey.hbm.xml 的源代码。<class>元素的 dynamic-update 属性设为 true, 这意味着当更新 MONKEYS 表时, Hibernate 会动态生成 SQL update 语句, 仅把需要更新的字段包含在 update 语句中。

例程 9-4 Monkey.hbm.xml

```
<hibernate-mapping>

  <class name="mypack.Monkey" table="MONKEYS" dynamic-update="true">
    <id name="id" type="long" column="ID">
      <generator class="increment" />
    </id>

    <property name="name" type="string" column="NAME" />

    <property name="homeAddress" type="mypack.AddressUserType" >
      <column name="HOME_PROVINCE" />
      <column name="HOME_CITY" />
      <column name="HOME_STREET" />
      <column name="HOME_ZIPCODE" />
    </property>

    <property name="comAddress" type="mypack.AddressUserType" >
      <column name="COM_PROVINCE" />
      <column name="COM_CITY" />
      <column name="COM_STREET" />
      <column name="COM_ZIPCODE" />
    </property>

    <property name="phone" type="mypack.PhoneUserType" column="PHONE" />

  </class>
</hibernate-mapping>
```

MONKEYS 表的 DDL 定义参见例程 9-5。

例程 9-5 MONKEYS 表的 DDL 定义

```
create table MONKEYS (
  ID bigint not null,
  NAME varchar(15),
  HOME_PROVINCE varchar(15),
  HOME_CITY varchar(15),
  HOME_STREET varchar(15),
  HOME_ZIPCODE varchar(6),
  COM_PROVINCE varchar(15),
  COM_CITY varchar(15),
  COM_STREET varchar(15),
  COM_ZIPCODE varchar(6),
  PHONE varchar(8),
  primary key (ID)
);
```

例程 9-6 是 Monkey 类的源程序。

例程 9-6 Monkey.java

```
package mypack;

public class Monkey{
    private Long id;
    private String name;
    private Address homeAddress;
    private Address comAddress;
    private Integer phone;

    public Monkey(String name, Address homeAddress, Address comAddress,
        Integer phone) {
        this.name = name;
        this.homeAddress = homeAddress;
        this.comAddress = comAddress;
        this.phone=phone;
    }
    public Monkey() { }

    //此处省略了 Monkey 的属性的 getXXX() 和 setXXX() 方法
    .....
}
```

Monkey 类的 homeAddress 属性为 Address 类型, 而不是 AddressUserType 类型:

```
AddressUserType homeAddress; //错误
Address homeAddress; //正确
```

Address 类是 homeAddress 属性的 Java 类型, 而 AddressUserType 类是 homeAddress 属性的 Hibernate 映射类型, 前者在 Monkey 类中定义, 后者在 Monkey.hbm.xml 文件中配置, 应该注意这两者的区别。

本章范例位于配套光盘的 sourcecode/chapter9 目录下。在 DOS 下转到本例的根目录 chapter9, 输入命令: ant run。该命令运行 BusinessService 类, 它的源程序参见例程 9-7。

例程 9-7 BusinessService.java

```
package mypack;

import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class BusinessService{
    public static SessionFactory sessionFactory;

    /** 初始化 Hibernate, 创建 SessionFactory 实例 */
    static{.....}
```

```

    /** 创建一个 Monkey 对象, 然后把它持久化 */
    public void saveMonkey(){.....}

    /** 创建一个 Address 对象, 然后把它持久化 */
    public void saveAddressSeparately(){.....}

    /** 更新 Monkey 对象, 修改 homeAddress、comAddress 和 name 属性 */
    public void updateMonkey(){.....}

    public void test(){
        saveMonkey();
        saveAddressSeparately();
        updateMonkey();
    }

    public static void main(String args[]){
        new BusinessService().test();
        sessionFactory.close();
    }
}

```

BusinessService 类的 main()方法调用 test()方法, test()方法又依次调用以下方法。

### 1. saveMonkey()方法

该方法先创建一个 Monkey 对象, 然后设置它的 name、homeAddress、comAddress 和 phone 属性,最后调用 session.save(monkey)方法持久化 Monkey 对象:

```

tx = session.beginTransaction();
Monkey monkey=new Monkey();
Address homeAddress=
    new Address("homeProvince","homeCity","homeStreet","100001");
Address comAddress=
    new Address("comProvince","comCity","comStreet","200002");
monkey.setName("Tom");
monkey.setHomeAddress(homeAddress);
monkey.setComAddress(comAddress);
monkey.setPhone(new Integer(55556666));

session.save(monkey);
tx.commit();

```

当 Hibernate 持久化 Monkey 对象时, 执行以下 insert 语句:

```

insert into MONKEYS (NAME, HOME_PROVINCE,
HOME_CITY, HOME_STREET, HOME_ZIPCODE, COM_PROVINCE, COM_CITY,
COM_STREET, COM_ZIPCODE, PHONE, ID)
values ('Tom', 'homeProvince', 'homeCity', 'homeStreet', '100001',
'comProvince', 'comCity', 'comStreet', '200002', '55556666',1);

```



## 2. saveAddressSeparately()方法

该方法试图单独持久化一个 Address 对象：

```
tx = session.beginTransaction();
Address homeAddress=new Address("homeProvince","homeCity",
    "homeStreet","100001");
session.save(homeAddress);
tx.commit();
```

由于 Address 是值类型，而非实体类型，因此不允许被单独持久化。执行该方法时，Hibernate 会抛出以下异常：

```
org.hibernate.MappingException: Unknown entity class: mypack.Address
```

关于值类型和实体类型的区别，参见本书第 8 章的 8.3.1 节（区分值类型和实体类型）。

## 3. updateMonkey()方法

该方法修改 Monkey 对象的 homeAddress 属性和 comAddress 属性：

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.load(Monkey.class,new Long(1));
Address homeAddress=new Address("homeProvince","homeCity",
    "homeStreet","100001");
Address comAddress=new Address("comProvinceNew","comCityNew",
    "comStreetNew","200002");
monkey.setHomeAddress(homeAddress);
monkey.setComAddress(comAddress);

tx.commit();
```

由于 Address 类是不可变类，因此必须重新创建两个 Address 实例，然后使 Monkey 对象的 homeAddress 属性和 comAddress 属性分别引用这两个实例。

当 Hibernate 清理缓存中的 Monkey 持久化对象时，会比较 Monkey 对象的属性及相应的快照是否相同，如果不同，就按照更新后的属性值来同步更新数据库。Hibernate 执行的 update 语句为：

```
update MONKEYS set COM_PROVINCE='comProvinceNew',
    COM_CITY='comCityNew', COM_STREET='comStreetNew',
    COM_ZIPCODE='200002'
where ID=1;
```

由于 Monkey.hbm.xml 文件中<class>元素的 dynamic-update 属性设为 true，因此更新 MONKEYS 表时，Hibernate 仅把需要更新的字段包含在 update 语句中。从以上 update 语句看出，Hibernate 没有修改 MONKEYS 表中和 homeAddress 属性对应的字段。

下面是运行 `updateMonkey()` 方法时, Hibernate 处理 Monkey 对象的 `homeAddress` 和 `comAddress` 属性的流程。

(1) 在通过 `session.load()` 方法加载 Monkey 对象时, Hibernate 调用 `AddressUserType` 类的 `deepCopy()` 方法生成 `homeAddress` 属性的快照:

```
public Object deepCopy(Object value) {
    return value;
}
```

以上 `deepCopy()` 方法直接返回代表 `homeAddress` 属性的 `value` 参数, 因此 `homeAddress` 属性和它的快照引用同一个 `Address` 对象, 参见图 9-3。

(2) Hibernate 接着调用 `AddressUserType` 类的 `deepCopy()` 方法生成 `comAddress` 属性的快照。和 `homeAddress` 属性同样, `comAddress` 属性和它的快照也引用同一个 `Address` 对象, 如图 9-3 所示。



图 9-3 `homeAddress` 属性和 `comAddress` 属性与它们的快照分别引用同一个 `Address` 对象

(3) 在应用程序中修改 Monkey 对象的 `homeAddress` 属性和 `comAddress` 属性, 使它们分别引用新的 `Address` 实例。这时, `homeAddress` 属性和它的快照不再引用同一个 `Address` 实例, 同样, `comAddress` 属性和它的快照也不再引用同一个实例, 参见图 9-4。

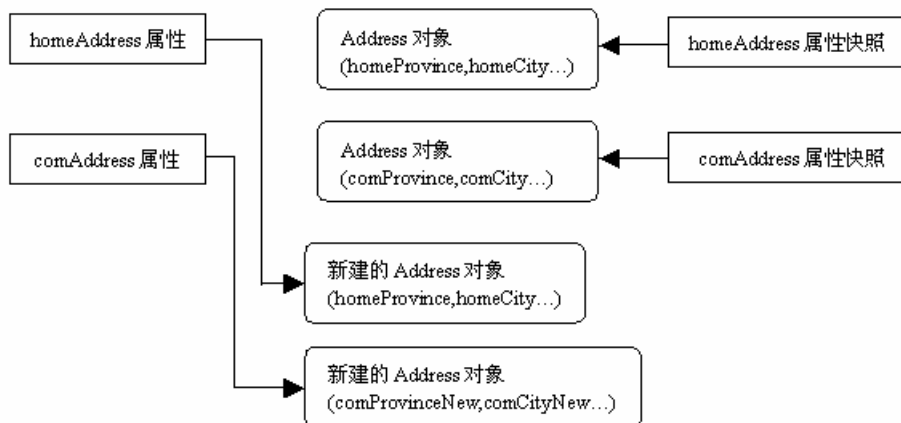


图 9-4 `homeAddress` 属性和 `comAddress` 属性与它们的快照分别引用不同的 `Address` 对象

(4) Hibernate 清理 Session 缓存中的 Monkey 对象时, 调用 AddressUserType 类的 equals()方法, 比较 Monkey 对象的 homeAddress 属性和 comAddress 属性是否与它们的快照相同。AddressUserType 类的 equals()方法的代码如下:

```
public boolean equals(Object x, Object y) {
    if (x == y) return true;
    if (x == null || y == null) return false;
    return x.equals(y);
}
```

#### Tips

AddressUserType 类的 equals()方法的两个参数 x 和 y 声明为 Object 类, 在运行时, 它们分别引用两个 Address 对象, 按照 Java 动态绑定的规则, JVM 会调用在 Address 类中定义的 equals()方法, 而不是 Object 类的 equals()方法。

AddressUserType 类的 equals()方法最后调用 Address 类的 equals()方法, 该方法比较两个 Address 对象的 province、city、street 和 zipcode 属性的字符串值是否相同, 比较结果为 homeAddress 属性与它的快照相同, comAddress 属性与它的快照不同。

(5) 由于 Monkey.hbm.xml 中<class>元素的 dynamic-update 属性为 true, 因此 Hibernate 在 update 语句中仅包含和 comAddress 属性对应的字段:

```
update MONKEYS set COM_PROVINCE='comProvinceNew',
COM_CITY='comCityNew' .....
```

## 9.5 小结

Hibernate 映射类型是 Java 类型和 SQL 类型之间的桥梁。Hibernate 映射类型分为两种: 内置映射类型和客户化映射类型。内置映射类型负责把一些常见的 Java 类型映射到相应的 SQL 类型; 此外, Hibernate 还允许用户实现 UserType 接口, 来灵活地定制客户化映射类型。

本章还提到了可变类和不可变类。所谓可变类, 就是指创建了其实例后, 可以修改它的属性; 与此相反, 则是不可变类。例如, 当 Address 类为可变类时, 可以直接修改一个 Address 对象的 province 属性:

```
//修改代表 Monkey 对象家庭地址的 Address 对象的 province 属性
Address address=monkey.getHomeAddress();
address.setProvince("Province2");
```

当 Address 类为不可变类时, 就无法修改一个 Address 对象的 province 属性。此时如果要修改一个 Monkey 对象的家庭地址中的省信息, 就必须使 Monkey 对象引用一个新的 Address 对象:

```
Address oldAddress=monkey.getHomeAddress();
Address newAddress=new Address("Province2",
                                oldAddress.getCity(),
                                oldAddress.getStreet(),
                                oldAddress.getZipcode());
monkey.setHomeAddress(newAddress);
```

在创建客户化映射类型时,deepCopy()方法用于生成持久化对象的属性的快照。当 Hibernate 清理缓存中的持久化对象时,会比较对象的属性及相应的快照是否相同,如果不同,就按照更新后的属性值来同步更新数据库。对于可变类型,deepCopy()方法返回属性值的备份,对于不可变类型,deepCopy()方法直接返回属性值。

例如当 Address 类为不可变类时,对应的 AddressUserType 类的 deepCopy()方法的实现如下:

```
public Object deepCopy(Object value) {
    return value;
}
```

当 Address 类为可变类时,对应的 AddressUserType 类的 deepCopy()方法的实现如下:

```
public Object deepCopy(Object value) {
    if (value == null) return null;
    Address address = (Address)value;
    return new Address(address.getProvince(),
                        address.getCity(),address.getStreet(),address.getZipcode());
}
```

## 第 10 章 映射继承关系

花果山的猴子（对应 `Monkey` 类）分为两种：金丝猴（对应 `JMonkey` 类）与长尾猴（对应 `CMonkey` 类）。`JMonkey` 类有一个表示毛色的 `color` 属性，`CMonkey` 类有一个表示尾巴长度的 `length` 属性。

在对象模型中，类与类之间除了关联关系和组成关系，还可以存在继承关系。例如 `JMonkey` 类与 `Monkey` 类，以及 `JMonkey` 类与 `Monkey` 类之间都是继承关系。

在图 10-1 所示的对象模型中，`Team` 类和 `Monkey` 类之间为一对多的双向关联关系。`Monkey` 类为抽象类，因此它不能被实例化，它有两个具体的子类：`JMonkey` 类和 `CMonkey` 类。`Monkey` 类、`JMonkey` 类和 `CMonkey` 构成了一棵继承关系树。

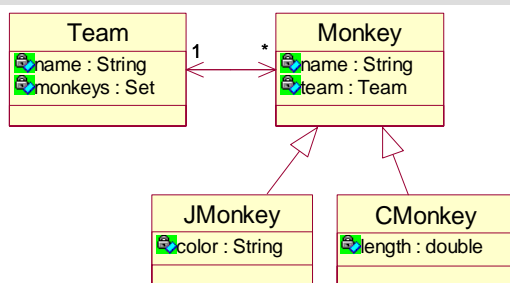


图 10-1 包含继承关系的对象模型

在面向对象的范畴中，还存在多态的概念，多态建立在继承关系的基础上。简单地理解，多态是指当一个 Java 应用变量被声明为 `Monkey` 类时，这个变量实际上既可以引用 `JMonkey` 类的实例，也可以引用 `CMonkey` 类的实例。以下这段程序代码就体现了多态：

```
List monkeys=businessService.findAllMonkeys();
Iterator it=monkeys.iterator();
while(it.hasNext()){
    Monkey m=(Monkey)it.next();
    if(m instanceof JMonkey)
        System.out.println(m.getName()+" "+((JMonkey)m).getColor());
    else
        System.out.println(m.getName()+" "+((CMonkey)m).getLength());
}
```

`BusinessService` 类的 `findAllMonkeys()` 方法通过 Hibernate API 从数据库中检索出所有

Monkey 对象。findAllMonkeys()方法返回的集合既包含 JMonkey 类的实例，也包含 CMonkey 类的实例，这种查询被称为多态查询。以上程序中变量 m 被声明为 Monkey 类型，它实际上既可能引用 JMonkey 类的实例，也可能引用 CMonkey 类的实例。

此外，从 Team 类到 Monkey 类为多态关联，因为 Team 类的 monkeys 集合中可以包含 JMonkey 类和 CMonkey 类的实例。从 Monkey 类到 Team 类不是多态关联，因为 Monkey 类的 team 属性只会引用 Team 类本身的实例。

数据库表之间并不存在继承关系，那么如何把对象模型的继承关系映射到关系数据模型中呢？本章将介绍以下 3 种映射方式：

- 继承关系树的每个具体类对应一个表：关系数据模型完全不支持对象模型中的继承关系和多态。
- 继承关系树的根类对应一个表：对关系数据模型进行非常规设计，在数据库表中加入额外的区分子类型的字段。通过这种方式，可以使关系数据模型支持继承关系和多态。
- 继承关系树的每个类对应一个表：在关系数据模型中用外键参照关系来表示继承关系。

#### Tips

具体类是指非抽象的类，具体类可以被实例化。JMonkey 类和 CMonkey 类就是具体类。

以上每种映射方式都有利有弊，本章除了介绍每种映射方式的具体步骤，还介绍它们的适用范围。

## 10.1 继承关系树的每个具体类对应一个表

把每个具体类映射到一张表是最简单的映射方式。如图 10-2 所示，在关系数据模型中只需定义 TEAMS、JMONKEYS 和 CMONKEYS 表。JMonkey 类和 JMONKEYS 表对应，JMonkey 类本身的 color 属性，以及从 Monkey 类中继承的 id 属性和 name 属性，在 JMONKEYS 表中都有对应的字段。此外，JMonkey 类继承了 Monkey 类与 Team 类的关联关系，与此对应，在 JMONKEYS 表中定义了参照 TEAMS 表的 TEAM\_ID 外键。

CMonkey 类和 CMONKEYS 表对应，CMonkey 类本身的 length 属性，以及从 Monkey 类中继承的 id 属性和 name 属性，在 CMONKEYS 表中都有对应的字段。此外，CMonkey 类继承了 Monkey 类与 Team 类的关联关系，与此对应，在 CMONKEYS 表中定义了参照 TEAMS 表的 TEAM\_ID 外键。

Team 类、JMonkey 类和 CMonkey 类都有相应的映射文件，而 Monkey 类没有

相应的映射文件。图 10-3 显示了持久化类、映射文件和数据库表之间的对应关系。

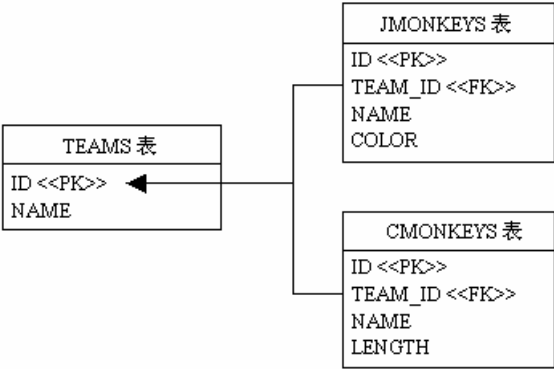


图 10-2 每个具体类对应一个表

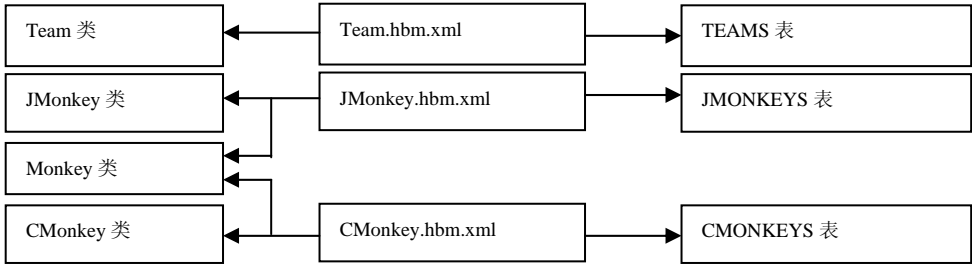


图 10-3 持久化类、映射文件和数据库表之间的对应关系

如果 **Monkey** 类不是抽象类，即 **Monkey** 类本身也能被实例化，那么还需要为 **Monkey** 类创建对应的 **MONKEYS** 表，此时 **JMONKEYS** 表和 **CMONKEYS** 表的结构仍然和图 10-2 中所示的一样。这意味着在 **MONKEYS** 表、**JMONKEYS** 表和 **CMONKEYS** 表中都定义了相同的 **NAME** 字段及参照 **TEAMS** 表的外键 **TEAM\_ID**。另外，还需为 **Monkey** 类创建单独的 **Monkey.hbm.xml** 文件。

10.1.1 创建映射文件

从 **Team** 类到 **Monkey** 类是多态关联，但是由于关系数据模型没有描述 **Monkey** 类和它的两个子类的继承关系，因此无法映射 **Team** 类的 **monkeys** 集合。例程 10-1 是 **Team.hbm.xml** 文件的代码，该文件仅映射了 **Team** 类的 **id** 和 **name** 属性。

例程 10-1 Team.hbm.xml

```
<hibernate-mapping >
  <class name="mypack.Team" table="TEAMS" >
    <id name="id" type="long" column="ID">
      <generator class="increment"/>
    </id>
```

```

        <property name="name" type="string" column="NAME" />

    </class>
</hibernate-mapping>

```

JMonkey.hbm.xml 文件用于把 JMonkey 类映射到 JMONKEYS 表，在这个映射文件中，除了需要映射 JMonkey 类本身的 color 属性，还需要映射从 Monkey 类中继承的 name 属性，此外还要映射从 Monkey 类中继承的与 Team 类的关联关系。例程 10-2 是 JMonkey.hbm.xml 文件的代码。

例程 10-2 JMonkey.hbm.xml

```

<hibernate-mapping >
    <class name="mypack.JMonkey" table="JMONKEYS">
        <id name="id" type="long" column="ID">
            <generator class="increment"/>
        </id>

        <property name="name" type="string" column="NAME" />
        <property name="color" type="String" column="COLOR" />

        <many-to-one
            name="team"
            column="TEAM_ID"
            class="mypack.Team"
        />
    </class>
</hibernate-mapping>

```

CMonkey.hbm.xml 文件用于把 CMonkey 类映射到 CMONKEYS 表，在这个映射文件中，除了需要映射 CMonkey 类本身的 length 属性，还需要映射从 Monkey 类中继承的 name 属性，此外还要映射从 Monkey 类中继承的与 Team 类的关联关系。例程 10-3 是 CMonkey.hbm.xml 文件的代码。

例程 10-3 CMonkey.hbm.xml

```

<hibernate-mapping >
    <class name="mypack.CMonkey" table="CMONKEYS">
        <id name="id" type="long" column="ID">
            <generator class="increment"/>
        </id>

        <property name="name" type="string" column="NAME" />
        <property name="length" type="double" column="LENGTH" />

        <many-to-one
            name="team"
            column="TEAM_ID"

```



```

        class="mypack.Team"
    />
</class>
</hibernate-mapping>

```

### 10.1.2 操纵持久化对象

操纵持久化对象映射方式不支持多态查询。对于以下查询语句：

```
List monkeys=session.createQuery("from Monkey").list();
```

如果 **Monkey** 类是抽象类，那么 **Hibernate** 会抛出异常。如果 **Monkey** 类是具体类，那么 **Hibernate** 仅查询 **MONKEYS** 表，检索出 **Monkey** 类本身的实例，但不会检索出它的两个子类的实例。本节的范例程序位于配套光盘的 `sourcecode\chapter10\10.1` 目录下，运行该程序前，需要在 **SAMPLEDB** 数据库中手工创建 **TEAMS** 表、**JMONKEYS** 表和 **CMONKEYS** 表，然后加入测试数据，相关的 SQL 脚本文件为 `10.1\schema\sampledb.sql`。

在 `chapter10` 目录下有 4 个 ANT 的工程文件，分别为 `build1.xml`、`build2.xml` 和 `build3.xml`，它们的区别在于文件开头设置的路径不一样，例如在 `build1.xml` 文件中设置了以下路径：

```

<property name="source.root" value="10.1/src"/>
<property name="class.root" value="10.1/classes"/>
<property name="lib.dir" value="lib"/>
<property name="schema.dir" value="10.1/schema"/>

```

在 DOS 命令行下进入 `chapter10` 根目录，然后输入命令：

```
ant -file build1.xml run
```

就会运行 **BusinessService** 类。ANT 命令的 `-file` 选项用于显式指定工程文件。**BusinessService** 类用于演示操纵 **Monkey** 类的对象的方法，例程 10-4 是它的源程序。

例程 10-4 `BusinessService.java`

```

public class BusinessService{
    public static SessionFactory sessionFactory;
    static{.....} /* 初始化 Hibernate, 创建 SessionFactory 实例 */

    public void saveMonkey(Monkey monkey){.....}
    public List findAllMonkeys(){.....}
    public Team loadTeam(long id){.....}

    public void test() throws Exception{
        List monkeys=findAllMonkeys();
        printAllMonkeys(monkeys.iterator());
    }
}

```

```

        Team team=loadTeam(1);
        printAllMonkeys(team.getMonkeys().iterator());

        Monkey monkey=new JMonkey("Mary","yellow",team);
        saveMonkey(monkey);
    }

    private void printAllMonkeys(Iterator it){
        while(it.hasNext()){
            Monkey m=(Monkey)it.next();
            if(m instanceof JMonkey)
                System.out.println(((JMonkey)m).getColor());
            else
                System.out.println(((CMonkey)m).getLength());
        }
    }

    public static void main(String args[]){
        new BusinessService().test();
        sessionFactory.close();
    }
}

```

BusinessService 的 main()方法调用 test()方法，test()方法依次调用以下方法。

- findAllMonkeys(): 检索数据库中所有的 Monkey 对象。
- loadTeam(): 加载一个 Team 对象。
- saveMonkey(): 保存一个 Monkey 对象。

(1) 运行 findAllMonkeys()方法，它的代码如下：

```

List results=new ArrayList();
tx = session.beginTransaction();
List jMonkeys=session.createQuery("from JMonkey").list();
results.addAll(jMonkeys);

List cMonkeys=session.createQuery("from CMonkey").list();
results.addAll(cMonkeys);

tx.commit();
return results;

```

为了检索所有的 Monkey 对象，必须分别检索所有的 JMonkey 实例和 CMonkey 实例，然后把它们合并到同一个集合中。在运行第一个 Query 的 list()方法时，Hibernate 执行以下 select 语句：

```
select * from JMONKEYS;
```

在运行第二个 Query 的 list()方法时，Hibernate 执行以下 select 语句：

```
select * from CMONKEYS;
```

(2) 运行 loadTeam()方法，它的代码如下：

```
tx = session.beginTransaction();
Team team=(Team)session.get(Team.class,new Long(id));

List jMonkeys=session
.createQuery("from JMonkey j where j.team.id="+id)
.list();
team.getMonkeys().addAll(jMonkeys);

List cMonkeys=session
.createQuery("from CMonkey c where c.team.id="+id)
.list();
team.getMonkeys().addAll(cMonkeys);

tx.commit();
return team;
```

由于这种映射方式不支持多态关联，因此由 Session 的 get()方法加载的 Team 对象的 monkeys 集合中不包含任何 Monkey 对象。BusinessService 类必须负责从数据库中检索出所有与 Team 对象关联的 JMonkey 对象及 CMonkey 对象，然后把它们加入到 monkeys 集合中。

(3) 运行 saveMonkey(Monkey monkey)方法，它的代码如下：

```
tx = session.beginTransaction();
session.save(monkey);
tx.commit();
```

在 test()方法中，创建了一个 JMonkey 实例，然后调用 saveMonkey()方法保存这个实例：

```
Monkey monkey=new JMonkey("Mary","yellow",team);
saveMonkey(monkey);
```

Session 的 save()方法能判断 monkey 变量实际引用的实例的类型，如果 monkey 变量引用 JMonkey 实例，就向 JMONKEYS 表插入一条记录，执行如下 insert 语句：

```
insert into JMONKEYS(ID,NAME,COLOR,TEAM_ID)
values(3, 'Mary', 'yellow',1);
```

如果 monkey 变量引用 CMonkey 实例，就向 CMONKEYS 表插入一条记录。

## 10.2 继承关系树的根类对应一个表

这种映射方式只需为继承关系树的 **Monkey** 根类创建一张表 **MONKEYS**。如图 10-4 所示，在 **MONKEYS** 表中不仅提供和 **Monkey** 类的属性对应的字段，还要提供和它的两个子类的所有属性对应的字段，此外，**MONKEYS** 表中需要额外加入一个字符串类型的 **MONKEY\_TYPE** 字段，用于区分 **Monkey** 的具体类型。

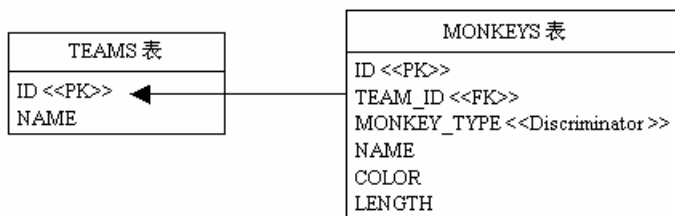


图 10-4 继承关系树的根类对应一个表

**Team** 类和 **Monkey** 类有相应的映射文件，而 **JMonkey** 类和 **CMonkey** 类没有相应的映射文件。图 10-5 显示了持久化类、映射文件和数据库表之间的对应关系。

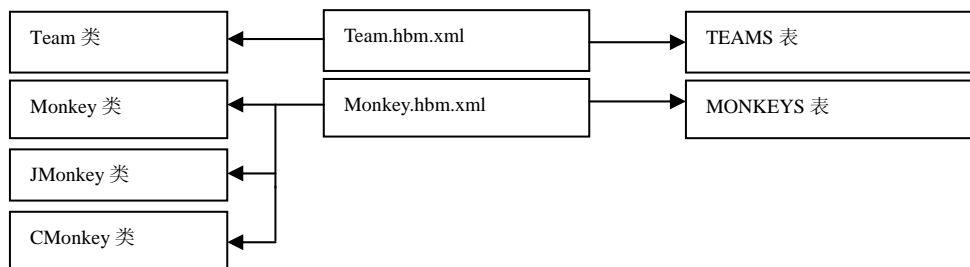


图 10-5 持久化类、映射文件和数据库表之间的对应关系

### 10.2.1 创建映射文件

从 **Team** 类到 **Monkey** 类是多态关联，由于关系数据模型可以反映出 **Monkey** 类和它的两个子类的继承关系，因此可以映射 **Team** 类的 **monkeys** 集合。例程 10-5 是 **Team.hbm.xml** 文件的代码，该文件不仅映射了 **Team** 类的 **id** 和 **name** 属性，还映射了它的 **monkeys** 集合。

例程 10-5 Team.hbm.xml

```
<hibernate-mapping>
  <class name="mypack.Team" table="TEAMS">
    <id name="id" type="long" column="ID">
      <generator class="increment"/>
    </id>
```

```

        <property name="name" type="string" column="NAME" />
        <set
            name="monkeys"
            inverse="true" >
            <key column="TEAM_ID" />
            <one-to-many class="mypack.Monkey" />
        </set>

    </class>
</hibernate-mapping>

```

Monkey.hbm.xml 文件用于把 Monkey 类映射到 MONKEYS 表，在这个映射文件中，除了需要映射 Monkey 类本身的属性，还需要在<subclass>元素中映射两个子类的属性。例程 10-6 是 Monkey.hbm.xml 文件的代码。

例程 10-6 Monkey.hbm.xml

```

<hibernate-mapping >
    <class name="mypack.Monkey" table="MONKEYS">
        <id name="id" type="long" column="ID">
            <generator class="increment"/>
        </id>
        <discriminator column="MONKEY_TYPE" type="string" />
        <property name="name" type="string" column="NAME" />

        <many-to-one
            name="team"
            column="TEAM_ID"
            class="mypack.Team"
        />

        <subclass name="mypack.JMonkey" discriminator-value="JM" >
            <property name="color" type="String" column="COLOR" />
        </subclass>

        <subclass name="mypack.CMonkey" discriminator-value="CM" >
            <property name="length" type="double" column="LENGTH" />
        </subclass>

    </class>

</hibernate-mapping>

```

在 Monkey.hbm.xml 文件中，<discriminator>元素指定 MONKEYS 表中用于区分 Monkey 类型的字段为 MONKEY\_TYPE，两个<subclass>元素用于映射 JMonkey 类和 CMonkey 类，<subclass>元素的 discriminator-value 属性指定 MONKEY\_TYPE

字段的取值。MONKEYS 表中有以下记录：

ID	NAME	MONKEY_TYPE	COLOR	LENGTH	TEAM_ID
1	Tom	JM	yellow	NULL	1
2	Mike	JM	orange	NULL	1
3	Jack	CM	NULL	1.2	1
4	Linda	CM	NULL	2.0	1

其中 ID 为 1 和 2 的记录的 MONKEY\_TYPE 字段的取值为“JM”，因此它们对应 JMonkey 类的实例，其中 ID 为 3 和 4 的记录的 MONKEY\_TYPE 字段的取值为“CM”，因此它们对应 CMonkey 类的实例。

这种映射方式要求 MONKEYS 表中和子类属性对应的字段允许为 null，例如 ID 为 1 和 2 的记录的 LENGTH 字段为 null，而 ID 为 3 和 4 的记录的 COLOR 字段为 null。如果业务需求规定 CMonkey 对象的 color 属性不允许为 null，显然无法在 MONKEYS 表中为 LENGTH 字段定义 not null 约束，可见这种映射方式无法保证关系数据模型的数据完整性。

如果 Monkey 类不是抽象类，即它本身也能被实例化，那么可以在<class>元素中定义它的 discriminator 值，形式如下：

```
<class name="mypack.Monkey" table="MONKEYS" discriminator-value="MM">
```

以上代码表明，如果 MONKEYS 表中一条记录的 MONKEY\_TYPE 字段的取值为“MM”，那么它对应 Monkey 类本身的实例。

## 10.2.2 操纵持久化对象

这种映射方式支持多态查询，对于以下查询语句：

```
List monkeys=session.createQuery("from Monkey").list();
```

Hibernate 会检索出所有的 JMonkey 对象和 CMonkey 对象。此外，也可以单独查询 Monkey 类的两个子类的实例，例如：

```
List jMonkeys=session.createQuery("from JMonkey").list();
```

本节的范例程序位于配套光盘的 sourcecode\chapter10\10.2 目录下，运行该程序前，需要在 SAMPLEDB 数据库中手工创建 TEAMS 表和 MONKEYS 表，然后加入测试数据，相关的 SQL 脚本文件为\10.2\schema\sampladb.sql。

在 DOS 命令行下进入 chapter10 根目录，然后输入命令：

```
ant -file build2.xml run
```

就会运行 `BusinessService` 类。`BusinessService` 的 `main()`方法调用 `test()`方法，`test()`方法依次调用以下方法：

- `findAllJMonkeys()`：检索数据库中所有的 `JMonkey` 对象。
- `findAllMonkeys()`：检索数据库中所有的 `Monkey` 对象。
- `loadTeam()`：加载一个 `Team` 对象。
- `saveMonkey()`：保存一个 `Monkey` 对象。

(1) 运行 `findAllJMonkeys()`方法，它的代码如下：

```
tx = session.beginTransaction();
List results=session.createQuery("from JMonkey").list();
tx.commit();
return results;
```

在运行 `Query` 的 `list()`方法时，`Hibernate` 执行以下 `select` 语句：

```
select * from MONKEYS where MONKEY_TYPE='JM' ;
```

(2) 运行 `findAllMonkeys()`方法，它的代码如下：

```
tx = session.beginTransaction();
List results=session.createQuery("from Monkey").list();
tx.commit();
return results;
```

在运行 `Query` 的 `list()`方法时，`Hibernate` 执行以下 `select` 语句：

```
select * from MONKEYS;
```

在这种映射方式下，`Hibernate` 支持多态查询，对于从 `MONKEYS` 表获得的查询结果，如果 `MONKEY_TYPE` 字段取值为“JM”，就创建 `JMonkey` 实例，如果 `MONKEY_TYPE` 字段取值为“CM”，就创建 `CMonkey` 实例。

(3) 运行 `loadTeam()`方法，它的代码如下：

```
tx = session.beginTransaction();
Team team=(Team)session.get(Team.class,new Long(id));
Hibernate.initialize(team.getMonkeys());
tx.commit();
```

这种映射方式支持多态关联。如果在 `Team.hbm.xml` 文件中对 `monkeys` 集合设置了立即检索策略，那么 `Session` 的 `get()`方法加载的 `Team` 对象的 `monkeys` 集合中包含所有关联的 `Monkey` 对象。由于本书提供的 `Team.hbm.xml` 文件对 `monkeys` 集合采用默认的延迟检索策略，因此以上程序代码通过 `Hibernate` 类的静态 `initialize()`方法来显式初始化 `monkeys` 集合。

(4) 运行 `saveMonkey(Monkey monkey)`方法，它的代码如下：

```
tx = session.beginTransaction();
session.save(monkey);
tx.commit();
```

在 `test()` 方法中，创建了一个 `JMonkey` 实例，然后调用 `saveMonkey()` 方法保存这个实例：

```
Monkey monkey=new JMonkey("Mary","yellow",team);
saveMonkey(monkey);
```

`Session` 的 `save()` 方法能判断 `monkey` 变量实际引用的实例的类型，如果 `monkey` 变量引用 `JMonkey` 实例，就执行如下 `insert` 语句：

```
insert into MONKEYS(ID,NAME,COLOR,MONKEY_TYPE,TEAM_ID)
values(5, 'Mary', 'yellow', 'JM',1);
```

以上 `insert` 语句没有为 `CMonkey` 类的 `length` 属性对应的 `LENGTH` 字段赋值，因此这条记录的 `LENGTH` 字段为 `null`。

## 10.3 继承关系树的每个类对应一个表

在这种映射方式下，继承关系树的每个类及接口都对应一个表。在本例中，需要创建 `MONKEYS`、`JMONKEYS` 和 `CMONKEYS` 表。如图 10-6 所示，`MONKEYS` 表仅包含和 `Monkey` 类的属性对应的字段，`JMONKEYS` 表仅包含和 `JMonkey` 类的属性对应的字段，`CMONKEYS` 表仅包含和 `CMonkey` 类的属性对应的字段。此外，`JMONKEYS` 表和 `CMONKEYS` 表都以 `MONKEY_ID` 字段作为主键，该字段还同时作为外键参照 `MONKEYS` 表。

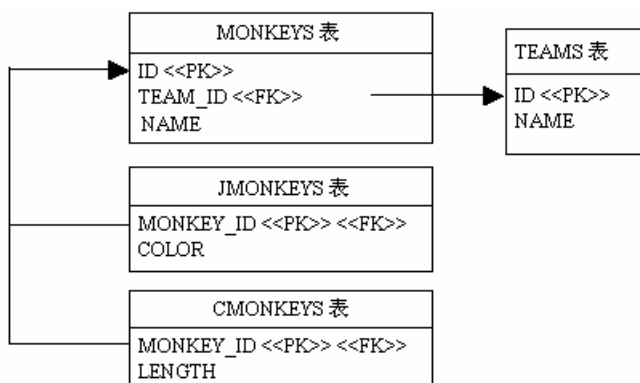


图 10-6 继承关系树的每个类对应一个表

`Team` 类和 `Monkey` 类有相应的映射文件，而 `JMonkey` 类和 `CMonkey` 类没有相应的映射文件。图 10-7 显示了持久化类、映射文件和数据库表之间的对应关系。



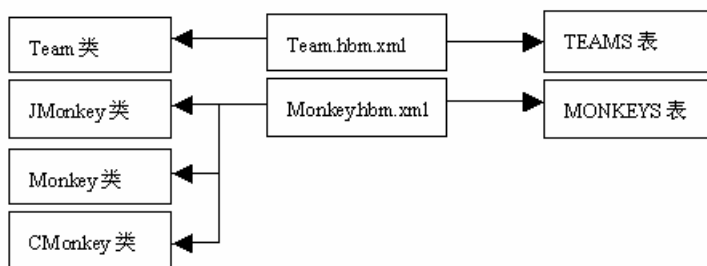


图 10-7 持久化类、映射文件和数据库表之间的对应关系

### 10.3.1 创建映射文件

从 Team 类到 Monkey 类是多态关联，由于关系数据模型反映出了 Monkey 类和它的两个子类的继承关系，因此可以映射 Team 类的 monkeys 集合。例程 10-7 是 Team.hbm.xml 文件的代码，该文件不仅映射了 Team 类的 id 和 name 属性，还映射了它的 monkeys 集合。

例程 10-7 Team.hbm.xml

```

<hibernate-mapping>
  <class name="mypack.Team" table="TEAMS">
    <id name="id" type="long" column="ID">
      <generator class="increment"/>
    </id>

    <property name="name" type="string" column="NAME" />
    <set
      name="monkeys"
      inverse="true">
        <key column="TEAM_ID" />
        <one-to-many class="mypack.Monkey" />
      </set>
    </class>
</hibernate-mapping>

```

Monkey.hbm.xml 文件用于把 Monkey 类映射到 MONKEYS 表，在这个映射文件中，除了需要映射 Monkey 类本身的属性，还需要在<joined-subclass>元素中映射两个子类的属性。例程 10-8 是 Monkey.hbm.xml 文件的代码。

例程 10-8 Monkey.hbm.xml

```

<hibernate-mapping>

  <class name="mypack.Monkey" table="MONKEYS">
    <id name="id" type="long" column="ID">
      <generator class="increment"/>

```

```

        </id>
        <property name="name" type="string" column="NAME" />

        <many-to-one
            name="team"
            column="TEAM_ID"
            class="mypack.Team"
        />

        <joined-subclass name="mypack.JMonkey" table="JMONKEYS" >
            <key column="MONKEY_ID" />
            <property name="color" type="String" column="COLOR" />
        </joined-subclass>

        <joined-subclass name="mypack.CMonkey" table="CMONKEYS" >
            <key column="MONKEY_ID" />
            <property name="length" type="double" column="LENGTH" />
        </joined-subclass>

    </class>
</hibernate-mapping>

```

在 Monkey.hbm.xml 文件中，两个<joined-subclass>元素用于映射 JMonkey 类和 CMonkey 类，<joined-subclass>元素的<key>子元素指定 JMONKEYS 表和 CMONKEYS 表中既作为主键又作为外键的 MONKEY\_ID 字段。图 10-8 显示了 MONKEYS 表、JMONKEYS 表和 CMONKEYS 表中记录的参照关系。



图 10-8 MONKEYS 表、JMONKEYS 表和 CMONKEYS 表中记录的参照关系

也可以在单独的映射文件中配置<subclass>或<joined-subclass>元素，但此时必须显式设定它们的 extends 属性。例如可以在单独的 JMonkey.hbm.xml 文件中映射 JMonkey 类：

```

<hibernate-mapping>
    <joined-subclass
        name="mypack.JMonkey"
        table="JMONKEYS"
        extends="mypack.Monkey" >
        .....

```

```
</joined-class>
</hibernate-mapping>
```

### 10.3.2 操纵持久化对象

这种映射方式支持多态查询，对于以下查询语句：

```
List monkeys=session.createQuery("from Monkey").list();
```

Hibernate 会检索出所有的 JMonkey 对象和 CMonkey 对象。此外，也可以单独查询 Monkey 类的两个子类的实例，例如：

```
List jMonkeys=session.createQuery("from JMonkey").list();
```

本节的范例程序位于配套光盘的 sourcecode\chapter10\10.3 目录下，运行该程序前，需要在 SAMPLEDB 数据库中手工创建 TEAMS 表、MONKEYS 表、JMONKEYS 表和 CMONKEYS 表，然后加入测试数据，相关的 SQL 脚本文件为 10.3\schema\sampldb.sql。

在 DOS 命令行下进入 chapter10 根目录，然后输入命令：

```
ant -file build3.xml run
```

就会运行 BusinessService 类。BusinessService 的 main() 方法调用 test() 方法，test() 方法依次调用以下方法：

- findAllJMonkeys(): 检索数据库中所有的 JMonkey 对象。
- findAllMonkeys(): 检索数据库中所有的 Monkey 对象。
- loadTeam(): 加载一个 Team 对象。
- saveMonkey(): 保存一个 Monkey 对象。

(1) 运行 findAllJMonkeys() 方法，它的代码如下：

```
tx = session.beginTransaction();
List results=session.createQuery("from JMonkey").list();
tx.commit();
return results;
```

在运行 Query 的 list() 方法时，Hibernate 执行以下 select 语句：

```
select * from JMONKEYS jm inner join MONKEYS m
on jm.MONKEY_ID=m.ID;
```

Hibernate 通过 JMONKEYS 表与 MONKEYS 表的内连接获得 JMonkey 对象的所有属性值。

(2) 运行 findAllMonkeys() 方法，它的代码如下：

```
tx = session.beginTransaction();
List results=session.createQuery("from Monkey").list();
```

```
tx.commit();
return results;
```

在运行 Query 的 list()方法时，Hibernate 执行以下 select 语句：

```
select * from MONKEYS m
left outer join JMONKEYS jm on m.ID=jm.MONKEY_ID
left outer join CMONKEYS cm on m.ID=cm.MONKEY_ID;
```

Hibernate 把 MONKEYS 表与 JMONKEYS 表及 CMONKEYS 表进行左外连接，从而获得 JMonkey 对象和 CMonkey 对象的所有属性值。在这种映射方式下，Hibernate 支持多态查询，对于以上查询语句获得的查询结果，如果 JMONKEYS 表的 MONKEY\_ID 字段不为 null，就创建 JMonkey 实例，如果 CMONKEYS 表的 MONKEY\_ID 字段不为 null，就创建 CMonkey 实例。

(3) 运行 loadTeam()方法，它的代码如下：

```
tx = session.beginTransaction();
Team team=(Team)session.get(Team.class,new Long(id));
Hibernate.initialize(team.getMonkeys());
tx.commit();
```

这种映射方式支持多态关联。如果在 Team.hbm.xml 文件中对 monkeys 集合设置了立即检索策略，那么 Session 的 get()方法加载的 Team 对象的 monkeys 集合中包含所有关联的 Monkey 对象。由于本书提供的 Team.hbm.xml 文件对 monkeys 集合使用了默认的延迟检索策略，因此，以上程序代码还通过 Hibernate 类的静态 initialize()方法来显式初始化 monkeys 集合，初始化时执行以下 select 语句：

```
select m.TEAM_ID, m.ID,m.NAME, m.TEAM_ID,
jm.COLOR, cm.LENGTH,
case when jm.MONKEY_ID is not null then 1
when cm.MONKEY_ID is not null then 2
when m.ID is not null then 0 end as clazz
from MONKEYS m left outer join JMONKEYS jm on m.ID=jm.MONKEY_ID
left outer join CMONKEYS cm on m.ID=cm.MONKEY_ID
where m.TEAM_ID=1
```

(4) 运行 saveMonkey(Monkey monkey)方法，它的代码如下：

```
tx = session.beginTransaction();
session.save(monkey);
tx.commit();
```

在 test()方法中，创建了一个 JMonkey 实例，然后调用 saveMonkey()方法保存这个实例：

```
Monkey monkey=new JMonkey("Mary","yellow",team);
saveMonkey(monkey);
```

Session 的 save()方法能判断 monkey 变量实际引用的实例的类型,如果 monkey 变量引用 JMonkey 实例,就执行如下 insert 语句:

```
insert into MONKEYS (ID,NAME, TEAM_ID) values (5, 'Mary', 1);
insert into JMONKEYS (MONKEY_ID ,COLOR) values (5, "yellow");
```

可见,每保存一个 JMonkey 对象,需要分别向 MONKEYS 表和 JMONKEYS 表插入一条记录,MONKEYS 表的记录和 JMONKEYS 表的记录共享同一个主键。

### 10.4 选择继承关系的映射方式

本章介绍的 3 种映射方式各有优缺点,表 10-1 对这 3 种映射方式做了比较。

表 10-1 比较 3 种映射方式			
比较方面	每个具体类对应一个表	根类对应一个表	每个类对应一个表
关系数据模型的复杂度	缺点: 每个具体类对应一个表, 这些表中包含重复字段	优点: 只需创建一个表	缺点: 表的数目最多, 并且表之间还有外键参照关系
查询性能	缺点: 如果要查询父类的对象, 必须查询所有具体的子类对应的表	优点: 有很好的查询性能, 无须进行表的连接	缺点: 需要进行表的内连接或左外连接
数据库 Schema 的可维护性	缺点: 如果父类的属性发生变化, 必须修改所有具体的子类对应的表	优点: 只需修改一张表	优点: 如果某个类的属性发生变化, 只需修改和这个类对应的表
是否支持多态查询和多态关联	缺点: 不支持	优点: 支持	优点: 支持
是否符合关系数据模型的常规设计规则	优点: 符合	缺点: (1) 在表中引入额外的区分子类的类型的字段 (2) 如果子类中的某个属性不允许为 null, 在表中无法为对应的字段创建 not null 约束	优点: 符合

总之, 选择继承关系的映射方式可以遵循以下原则:

- 如果不需要支持多态查询和多态关联, 可以采用每个具体类对应一个表的映射方式。
- 如果需要支持多态查询和多态关联, 并且子类包含的属性不多, 可以采用根类对应一个表的映射方式。
- 如果需要支持多态查询和多态关联, 并且子类包含的属性很多, 可以采用每个类对应一个表的映射方式。
- 如果继承关系树中包含接口, 可以把它当做抽象类来处理。

## 10.5 小结

本章介绍了映射继承关系的 3 种方式：

- 继承关系树的每个具体类对应一个表：在具体类对应的表中，不仅包含和具体类的属性对应的字段，还包含和具体类的父类的属性对应的字段。这种映射方式不支持多态关联和多态查询。
- 继承关系树的根类对应一个表：在根类对应的表中，不仅包含和根类的属性对应的字段，还包含和所有子类的属性对应的字段。这种映射方式支持多态关联和多态查询，并且能获得最佳查询性能，缺点是需要对关系数据模型进行非常规设计，在数据库表中加入额外的区分各个子类的字段，此外，不能为所有子类的属性对应的字段定义 `not null` 约束。
- 继承关系树的每个类对应一个表：在每个类对应的表中只需包含和这个类本身的属性对应的字段，子类对应的表参照父类对应的表。这种映射方式支持多态关联和多态查询，而且符合关系数据模型的常规设计规则，缺点是它的查询性能不如第二种映射方式。在这种映射方式下，必须通过表的内连接或左外连接来实现多态查询和多态关联。

在默认情况下，对于简单的继承关系树可以采用根类对应一个表的映射方式。如果必须保证关系数据模型的数据完整性，可以采用每个类对应一个表的映射方式。对于复杂的继承关系树，可以将它分解为几棵子树，对每棵子树采用不同的映射方式。当然，在设计对象模型时，应该尽量避免设计过分复杂的继承关系，这不仅会增加把对象模型映射到关系数据模型的难度，而且也会增加在 Java 程序代码中操纵持久化对象的复杂度。

对于不同的映射方式，必须创建不同的关系数据模型和映射文件，但是对象模型是一样的，对象模型中的持久化类的实现也都一样。只要具备 Java 编程基础知识，就能创建具有继承关系的持久化类，因此本章没有详细介绍这些持久化类的创建过程，在此仅提醒一点，子类的完整构造方法不仅负责初始化子类本身的属性，还应该负责初始化从父类中继承的属性，如以下是 JMonkey 类的构造方法：

```
public class JMonkey extends Monkey{
    private String color;

    /** 完整构造方法*/
    public JMonkey(String name, String color,Team team) {
        super(name,team);
        this.color=color;
    }
}
```

```
/** 默认构造方法*/  
public JMonkey() {}  
.....  
}
```

Hibernate 只会访问持久化类的默认构造方法, 永远不会访问其他形式的构造方法。提供以上形式的完整构造方法, 主要是为 Java 应用的编程提供方便。

9

10

11

12

13

14

15

16

## 第 11 章 Java 集合

本书第 5 章已经讲过，一个武术队（对应 `Team` 类）包括许多猴子（对应 `Monkey` 类），为了描述这种关联关系，在 `Team` 类中定义了一个集合类型的属性 `monkeys`：

```
private Set monkeys=new HashSet();
```

以上 `Set` 接口属于 Java 集合的一种。本章将从运用的角度，介绍一些常用 Java 集合的特性和使用方法。本章主要是为第 12 章（映射值类型集合）做铺垫，因为只有深刻理解了 Java 集合的特性，才能进一步了解如何在 `Hibernate` 应用中映射 Java 集合。如果读者已经对 Java 集合非常熟悉，可以略过本章，直接阅读第 12 章（映射值类型集合）。

Java 的集合类都位于 `java.util` 包中，Java 集合中存放的是对象的引用，而非对象本身，出于表达上的便利，下文把“集合中的对象的引用”简称为“集合中的对象”。Java 集合主要分为 3 种类型：

- **Set（集）**：集合中的对象不按特定方式排序，并且没有重复对象。它的有些实现类能对集合中的对象按特定方式排序。
- **List（列表）**：集合中的对象按索引位置排序，可以有重复对象，允许按照对象在集合中的索引位置检索对象。
- **Map（映射）**：集合中的每一个元素包含一对键对象和值对象，集合中没有重复的键对象，值对象可以重复。它的有些实现类能对集合中的键对象进行排序。

### Tips

`Set`、`List` 和 `Map` 统称为 Java 集合，其中 `Set` 与数学中的集合最接近，两者都不允许包含重复元素。

图 11-1 显示了 Java 的主要集合类的类框图。



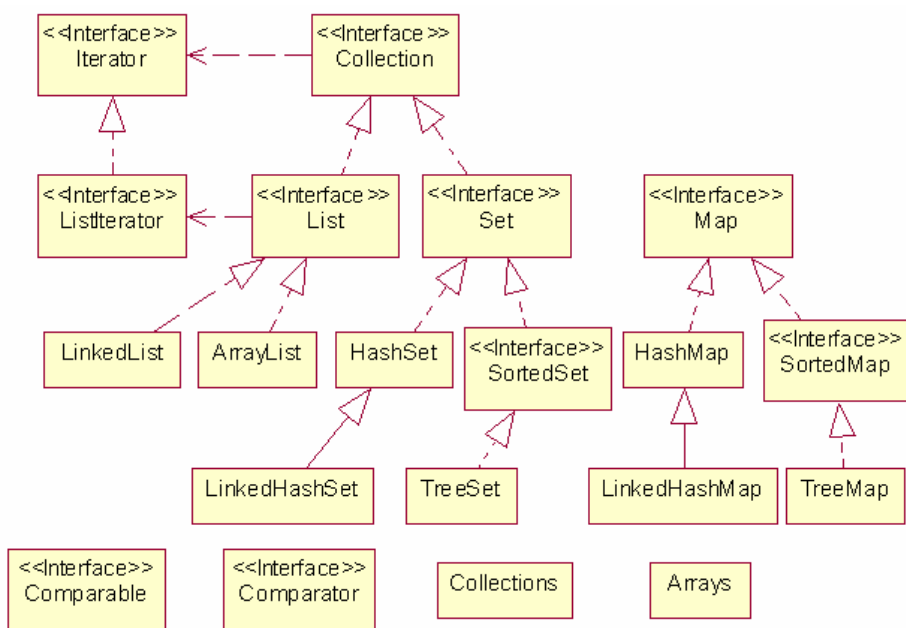


图 11-1 Java 的主要集合类的类框图

## 11.1 Set（集）

Set 是最简单的一种集合，集合中的对象不按特定方式排序，并且没有重复对象。Set 接口主要有两个实现类 HashSet 和 TreeSet。HashSet 类按照哈希算法来存取集合中的对象，存取速度比较快。HashSet 类还有一个子类 LinkedHashSet 类，它不仅实现了哈希算法，而且实现了链表数据结构。TreeSet 类实现了 SortedSet 接口，具有排序功能。

### 11.1.1 Set 的一般用法

Set 集合中存放的是对象的引用，并且没有重复对象。以下代码创建了 3 个引用变量 s1、s2 和 s3，s1 和 s2 变量引用同一个字符串对象“hello”，s3 变量引用另一个字符串对象“world”，Set 集合依次把这 3 个引用变量加入到集合中：

```

Set set=new HashSet();
String s1=new String("hello");
String s2=s1;
String s3=new String("world");
set.add(s1);
set.add(s2);
set.add(s3);
    
```

```
System.out.println(set.size()); //打印集合中对象的数目
```

以上程序的输出结果为 2, 实际上只向 Set 集合加入了两个对象, 如图 11-2 所示。

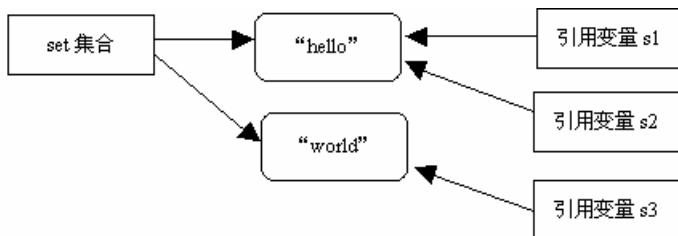


图 11-2 Set 集合中包含两个字符串对象

### Tips

本节程序选用 HashSet 作为 Set 实现类, 但是本节只涉及 Set 集合的基本特性, 这些特性不仅适用于 HashSet, 也适用于 TreeSet。

那么, 当一个新的对象加入到 Set 集合中时, Set 的 add() 方法是如何判断这个对象是否已经存在于集合中的呢? 下面这段代码演示了 add() 方法的判断流程, 其中 newObject 表示待加入的新对象:

```
boolean isExists=false;
Iterator it=set.iterator();
while(it.hasNext()){
    Object oldObject=it.next();
    if(newObject.equals(oldObject)){
        isExists=true;
        break;
    }
}
```

可见, Set 采用对象的 equals() 方法比较两个对象是否相等, 而不是采用 “==” 比较运算符。以下程序代码尽管两次调用了 Set 的 add() 方法, 实际上只加入了一个对象:

```
Set set=new HashSet();
String s1=new String("hello");
String s2=new String("hello");
set.add(s1);
set.add(s2);
System.out.println(set.size()); //打印集合中对象的数目
```

虽然变量 s1 和 s2 实际上引用的是两个内存地址不同的字符串对象, 但是由于 s2.equals(s1) 的比较结果为 true, 因此 Set 认为它们是相等的对象。当第二次调用 Set 的 add() 方法时, add() 方法不会把 s2 引用的字符串对象加入到集合中, 以上程序的输出结果为 1。

### 11.1.2 HashSet类

HashSet 类按照哈希算法来存取集合中的对象，具有很好的存取性能。当 HashSet 向集合中加入一个对象时，会调用对象的 hashCode()方法获得哈希码，然后根据这个哈希码进一步计算出对象在集合中的存放位置。

在 Object 类中定义了 hashCode()和 equals()方法，Object 类的 equals()方法按照内存地址比较对象是否相等，因此如果 object1.equals(object2)为 true，表明 object1 变量和 object2 变量实际上引用同一个对象，那么 object1 和 object2 的哈希码也肯定相同。

为了保证 HashSet 能正常工作，要求当两个对象用 equals()方法比较的结果为相等时，它们的哈希码也相等。也就是说，如果 monkey1.equals(monkey2)为 true，那么以下表达式的结果也为 true：

```
monkey1.hashCode() == monkey2.hashCode()
```

如果用户定义的 Monkey 类覆盖了 Object 类的 equals()方法，但是没有覆盖 Object 类的 hashCode()方法，就会导致当 monkey1.equals(monkey2)为 true 时，而 monkey1 和 monkey2 的哈希码不一定一样，这会使 HashSet 无法正常工作。

例程 11-1 是 Monkey 类的 equals()方法，它的比较规则为：如果两个 Monkey 对象的 name 属性和 age 属性相同，那么这两个 Monkey 对象相等。

例程 11-1 Monkey 类的 equals()方法

```
public boolean equals(Object o){
    if(this==o)return true;
    if(! (o instanceof Monkey)) return false;
    final Monkey other=(Monkey)o;

    if(this.name.equals(other.getName()) && this.age==other.getAge())
        return true;
    else
        return false;
}
```

以下程序向 HashSet 中加入两个 Monkey 对象：

```
Set set=new HashSet();
Monkey monkey1=new Monkey("Tom",15);
Monkey monkey2=new Monkey("Tom",15);
set.add(monkey1);
set.add(monkey2);
System.out.println(set.size());
```

由于 monkey1.equals(monkey2)的比较结果为 true，按理说 HashSet 只应该把

monkey1 加入集合中，但实际上以上程序的输出结果为 2，表明集合中加入了两个对象。出现这一非正常现象的原因在于 monkey1 和 monkey2 的哈希码不一样，因此 HashSet 为 monkey1 和 monkey2 计算出不同的存放位置，于是把它们存放在集合中的不同地方。

可见，为了保证 HashSet 正常工作，如果 Monkey 类覆盖了 equals() 方法，也应该覆盖 hashCode() 方法，并且保证两个相等的 Monkey 对象的哈希码也一样，与例程 11-1 的 equals() 方法对应，可按以下方式定义 Monkey 类的 hashCode() 方法：

```
public int hashCode(){
    int result;
    result=name.hashCode();
    result = 29 * result + age;
    return result;
}
```

由于在 equals() 方法中按 Monkey 类的 name 属性和 age 属性比较两个 Monkey 对象是否相等，因此在 hashCode() 方法中也只需要包含 name 属性和 age 属性。以上程序代码假定 name 属性不会为 null，因此没有先判断 name 属性是否为 null，就直接调用 name 属性的 hashCode() 方法。本例中的 age 属性为 int 类型，如果 age 属性为 Integer 类型，那么可按以下方式定义 hashCode() 方法：

```
public int hashCode(){
    int result;
    result= (name==null?0:name.hashCode());
    result = 29 * result + (age==null?0:age.hashCode());
    return result;
}
```

以上程序假定 name 属性和 age 属性都有可能为 null，因此为了保证程序代码的健壮性，先判断 name 和 age 属性是否为 null。

### 11.1.3 TreeSet 类

TreeSet 类实现了 SortedSet 接口，能够对集合中的对象进行排序。以下程序创建了一个 TreeSet 对象，然后向集合中加入 4 个 Integer 对象：

```
Set set=new TreeSet();
set.add(new Integer(8));
set.add(new Integer(7));
set.add(new Integer(6));
set.add(new Integer(9));

Iterator it=set.iterator();
while(it.hasNext()){
```

```
System.out.println(it.next());
}
```

以上程序的输出结果为：

```
6 7 8 9
```

当 `TreeSet` 向集合中加入一个对象时，会把它插入到有序的对象序列中。那么 `TreeSet` 是如何对对象进行排序的呢？`TreeSet` 支持两种排序方式：自然排序和客户化排序，在默认情况下 `TreeSet` 采用自然排序方式。

### 1. 自然排序

在 JDK 中，有一部分类实现了 `Comparable` 接口，如 `Integer`、`Double` 和 `String` 等。`Comparable` 接口有一个 `compareTo(Object o)` 方法，它返回整数类型。对于表达式 `x.compareTo(y)`，如果返回值为 0，表示 `x` 和 `y` 相等，如果返回值大于 0，表示 `x` 大于 `y`，如果返回值小于 0，表示 `x < y`。

`TreeSet` 调用对象的 `compareTo()` 方法比较集合中对象的大小，然后进行升序排列，这种排序方式称为自然排序。表 11-1 显示了 JDK 中实现了 `Comparable` 接口的一些类的排序方式。

表 11-1 JDK 中实现了 `Comparable` 接口的一些类的排序方式

类	排 序
<code>BigDecimal</code> 、 <code>BigInteger</code> 、 <code>Byte</code> 、 <code>Double</code> 、 <code>Float</code> 、 <code>Integer</code> 、 <code>Long</code> 、 <code>Short</code>	按数字大小排序
<code>Character</code>	按字符的 Unicode 值的数字大小排序
<code>String</code>	按字符串中字符的 Unicode 值排序

使用自然排序时，只能向 `TreeSet` 集合中加入同类型的对象，并且这些对象的类必须实现了 `Comparable` 接口。以下程序先后向 `TreeSet` 集合加入一个 `Integer` 对象和 `String` 对象：

```
Set set=new TreeSet();
set.add(new Integer(8));
set.add(new String("9")); //抛出 ClassCastException
```

当第二次调用 `TreeSet` 的 `add()` 方法时会抛出 `ClassCastException`：

```
java.lang.ClassCastException: java.lang.Integer
at java.lang.String.compareTo(String.java:825)
at java.util.TreeMap.compare(TreeMap.java:1047)
at java.util.TreeMap.put(TreeMap.java:449)
at java.util.TreeSet.add(TreeSet.java:198)
```

在 `String` 类的 `compareTo(Object o)` 方法中，首先对参数 `o` 进行类型转换：

```
String s=(String)o;
```

如果参数 `o` 实际引用的不是 `String` 类型的对象，以上代码就会抛出 `ClassCastException`。例程 11-2 向 `TreeSet` 集合加入 3 个 `Monkey` 对象，但是 `Monkey` 类没有实现 `Comparable` 接口。

例程 11-2 向 `TreeSet` 集合加入 3 个 `Monkey` 对象

```
Set set=new TreeSet();
set.add(new Monkey("Tom",15));
set.add(new Monkey("Tom",20));
set.add(new Monkey("Tom",15));
set.add(new Monkey("Mike",15));

Iterator it=set.iterator();
while(it.hasNext()){
    Monkey monkey=(Monkey)it.next();
    System.out.println(monkey.getName()+" "+monkey.getAge());
}
```

当第二次调用 `TreeSet` 的 `add()` 方法时，也会抛出 `ClassCastException` 异常。如果希望避免这种异常，应该使 `Monkey` 类实现 `Comparable` 接口；相应地，在 `Monkey` 类中应该实现 `compareTo()` 方法。以下是 `Monkey` 类的 `compareTo()` 方法的一种实现方式：

```
public int compareTo(Object o){
    Monkey other=(Monkey)o;

    //先按照 name 属性排序
    if(this.name.compareTo(other.getName())>0)return 1;
    if(this.name.compareTo(other.getName())<0)return -1;

    //再按照 age 属性排序
    if(this.age>other.getAge())return 1;
    if(this.age<other.getAge())return -1;
    return 0;
}
```

为了保证 `TreeSet` 能正确地排序，要求 `Monkey` 类的 `compareTo()` 方法与 `equals()` 方法按相同的规则比较两个 `Monkey` 对象是否相等。也就是说，如果 `monkey1.equals(monkey2)` 为 `true`，那么 `monkey1.compareTo(monkey2)` 为 0。

以上 `compareTo()` 方法判断两个 `Monkey` 对象相等的条件为 `name` 属性和 `age` 属性都相等，因此在 `Monkey` 类的 `equals()` 方法中应该采用相同的比较规则：

```
public boolean equals(Object o){
    if(this==o)return true;
    if(!(o instanceof Monkey)) return false;
    final Monkey other=(Monkey)o;
```

```

        if(this.name.equals(other.getName()) && this.age==other.getAge())
            return true;
        else
            return false;
    }

```

在本章 11.1.2 节已经指出，如果一个类重新实现了 `equals()` 方法，那么也应该重新实现 `hashCode()` 方法，并且保证当两个对象相等时，它们的哈希码相同，所以在 `Monkey` 类中还应该实现 `hashCode()` 方法：

```

    public int hashCode() {
        int result;
        result= (name==null?0:name.hashCode());
        result = 29 * result + age;
        return result;
    }

```

如果在 `Monkey` 类中实现了 `compareTo()`、`equals()` 和 `hashCode()` 方法，例程 11-2 的输出结果为：

```

Mike 15
Tom 15
Tom 20

```

值得注意的是，对于 `TreeSet` 中已经存在的 `Monkey` 对象，如果修改了它们的 `name` 属性或 `age` 属性，`TreeSet` 不会对集合进行重新排序，如以下程序先后把 `monkey1` 和 `monkey2` 对象加入到 `TreeSet` 集合中，然后修改 `monkey1` 的 `age` 属性：

```

Set set=new TreeSet();
Monkey monkey1=new Monkey("Tom",15);
Monkey monkey2=new Monkey("Tom",16);
set.add(monkey1);
set.add(monkey2);
monkey1.setAge(20);

Iterator it=set.iterator();
while(it.hasNext()){
    Monkey monkey=(Monkey)it.next();
    System.out.println(monkey.getName()+" "+monkey.getAge());
}

```

以上程序的输出结果为：

```

Tom 20
Tom 16

```

可见，当外部程序修改了 `monkey1` 对象的 `age` 属性后，`TreeSet` 不会重新排序。

在实际对象模型中，**Monkey** 类是实体类，**Monkey** 对象的 **name** 属性和 **age** 属性可以被更新，因此不适合通过 **TreeSet** 来排序。最适合于排序的是不可变类，本书第 9 章的 9.3 节（用客户化映射类型取代 **Hibernate** 组件）介绍了不可变类的概念，不可变类的主要特征是它的对象的属性不能被修改。

## 2. 客户化排序

除了自然排序，**TreeSet** 还支持客户化排序。**java.util.Comparator** 接口用于指定具体的排序方式，它有个 **compare(Object object1, Object object2)** 方法，用于比较两个对象的大小。当表达式 **compare(x,y)** 的值大于 0，表示 **x** 大于 **y**；当 **compare(x,y)** 的值小于 0，表示 **x** 小于 **y**；当 **compare(x,y)** 的值等于 0，表示 **x** 等于 **y**。如果希望 **TreeSet** 仅按照 **Monkey** 对象的 **name** 属性进行降序排列，可以先创建一个实现 **Comparator** 接口的类 **MonkeyComparator**，参见例程 11-3。

例程 11-3 **MonkeyComparator.java**

```
package mypack;
import java.util.*;

public class MonkeyComparator implements Comparator{
    public int compare(Object o1, Object o2){
        Monkey c1=(Monkey)o1;
        Monkey c2=(Monkey)o2;

        if(c1.getName().compareTo(c2.getName())>0) return -1;
        if(c1.getName().compareTo(c2.getName())<0) return 1;

        return 0;
    }
}
```

接下来在构造 **TreeSet** 的实例时，调用它的 **TreeSet(Comparator comparator)** 构造方法：

```
Set set=new TreeSet(new MonkeyComparator());
Monkey monkey1=new Monkey("Tom",15);
Monkey monkey3=new Monkey("Jack",16);
Monkey monkey2=new Monkey("Mike",26);
set.add(monkey1);
set.add(monkey2);
set.add(monkey3);

Iterator it=set.iterator();

while(it.hasNext()){
    Monkey monkey=(Monkey)it.next();
```



```
System.out.println(monkey.getName()+" "+monkey.getAge());
}
```

当 `TreeSet` 向集合中加入 `Monkey` 对象时，会调用 `MonkeyComparator` 类的 `compare()` 方法进行排序，以上 `TreeSet` 按照 `Monkey` 对象的 `name` 属性进行降序排列，最后输出的结果为：

```
Tom 15
Mike 26
Jack 16
```

## 11.2 List（列表）

`List` 的主要特征是其对象以线性方式存储，集合中允许存放重复对象。`List` 接口主要的实现类有 `LinkedList` 和 `ArrayList`。`LinkedList` 采用链表数据结构，而 `ArrayList` 代表大小可变的数组。

`List` 对集合中的对象按索引位置排序，允许按照对象在集合中的索引位置检索对象。以下程序向 `List` 中加入 4 个 `Integer` 对象：

```
List list=new ArrayList();
list.add(new Integer(3));
list.add(new Integer(4));
list.add(new Integer(3));
list.add(new Integer(2));
```

`List` 的 `get(int index)` 方法返回集合中由参数 `index` 指定的索引位置的对象，第一个加入到集合中的对象的索引位置为 0。以下程序依次检索出集合中的所有对象：

```
for(int i=0;i<list.size();i++)
    System.out.println(list.get(i));
```

以上程序的输出结果为：

```
3 4 3 2
```

`List` 的 `iterator()` 方法和 `Set` 的 `iterator()` 方法一样，也能返回 `Iterator` 对象，通过 `Iterator` 对象，可以遍历集合中的所有对象，例如：

```
Iterator it=list.iterator();
while(it.hasNext()){
    System.out.println(it.next());
}
```

`List` 只能对集合中的对象按索引位置排序，如果希望对 `List` 中的对象按其他特定方式排序，可以借助 `Comparator` 接口和 `Collections` 类。`Collections` 类是 Java 集合 API 中的辅助类，它提供了操纵集合的各种静态方法，其中 `sort()` 方法用于对 `List`

中的对象进行排序：

- `sort(List list)`：对 `List` 中的对象进行自然排序。
- `sort(List list,Comparator comparator)`：对 `List` 中的对象进行客户化排序，`comparator` 参数指定排序方式。

以下程序对 `List` 中的 `Integer` 对象进行自然排序：

```
List list=new ArrayList();
list.add(new Integer(3));
list.add(new Integer(4));
list.add(new Integer(3));
list.add(new Integer(2));

Collections.sort(list);
for(int i=0;i<list.size();i++)
    System.out.println(list.get(i));
```

以上程序的输出结果为：

```
2 3 3 4
```

## 11.3 Map（映射）

`Map`（映射）是一种把键对象和值对象进行映射的集合，它的每一个元素都包含一对键对象和值对象。向 `Map` 集合中加入元素时，必须提供一对键对象和值对象，从 `Map` 集合中检索元素时，只要给出键对象，就会返回对应的值对象，如下程序通过 `Map` 的 `put(Object key,Object value)` 方法向集合中加入元素，通过 `Map` 的 `get(Object key)` 方法来检索与键对象对应的值对象：

```
Map map=new HashMap();
map.put("1","Monday");
map.put("2","Tuesday");
map.put("3","Wednesday");
map.put("4","Thursday");
String day=(String)map.get("2"); //day 的值为“Tuesday”
```

`Map` 集合中的键对象不允许重复，也就是说，任意两个键对象通过 `equals()` 方法比较的结果都是 `false`。对于值对象则没有唯一性的要求，可以将任意多个键对象映射到同一个值对象上。例如，以下 `Map` 集合中的键对象“1”和“one”都和同一个值对象“Monday”对应：

```
Map map=new HashMap();
map.put("1","Mon.");
map.put("1","Monday");
map.put("one","Monday");
```

由于第一次和第二次加入 `Map` 中的键对象都为 “1”，因此第一次加入的值对象将被覆盖，`Map` 集合中最后只有两个元素，分别为：

```
“1” 对应 “Monday”
“one” 对应 “Monday”
```

`Map` 有两种比较常用的实现：`HashMap` 和 `TreeMap`。`HashMap` 按照哈希算法来存取键对象，有很好的存取性能，为了保证 `HashMap` 能正常工作，和 `HashSet` 一样，要求当两个键对象通过 `equals()` 方法比较为 `true` 时，这两个键对象的 `hashCode()` 方法返回的哈希码也一样。

`TreeMap` 实现了 `SortedMap` 接口，能对键对象进行排序。和 `TreeSet` 一样，`TreeMap` 也支持自然排序和客户化排序两种方式。以下程序中的 `TreeMap` 会对 4 个字符串类型的键对象 “1”、“3”、“4” 和 “2” 进行自然排序：

```
Map map=new TreeMap();
map.put("1","Monday");
map.put("3","Wendsday");
map.put("4","Thursday");
map.put("2","Tuesday");

Set keys=map.keySet();
Iterator it=keys.iterator();
while(it.hasNext()){
    String key=(String)it.next();
    String value=(String)map.get(key);
    System.out.println(key+" "+value);
}
```

`Map` 的 `keySet()` 方法返回集合中所有键对象的集合，以上程序的输出结果为：

```
1 Monday
2 Tuesday
3 Wendsday
4 Thursday
```

如果希望 `TreeMap` 进行客户化排序，可调用它的另一个构造方法 `TreeMap(Comparator comparator)`，参数 `comparator` 指定具体的排序方式。

## 11.4 小结

本章介绍了几种常用 Java 集合的特性和使用方法。为了保证集合正常工作，有些集合类对存放的对象有特殊的要求，归纳如下：

- **HashSet**：如果集合中对象所属的类重新定义了 `equals()` 方法，那么这个类

也必须重新定义 `hashCode()` 方法，并且保证当两个对象用 `equals()` 方法比较的结果为 `true` 时，这两个对象的 `hashCode()` 方法的返回值相等。

- **TreeSet**: 如果对集合中的对象进行自然排序，要求对象所属的类实现 `Comparable` 接口，并且保证这个类的 `compareTo()` 和 `equals()` 方法采用相同的比较规则来比较两个对象是否相等。
- **HashMap**: 如果集合中键对象所属的类重新定义了 `equals()` 方法，那么这个类也必须重新定义 `hashCode()` 方法，并且保证当两个键对象用 `equals()` 方法比较的结果为 `true` 时，这两个键对象的 `hashCode()` 方法的返回值相等。
- **TreeMap**: 如果对集合中的键对象进行自然排序，要求键对象所属的类实现 `Comparable` 接口，并且保证这个类的 `compareTo()` 和 `equals()` 方法采用相同的比较规则来比较两个键对象是否相等。

由此可见，为了使应用程序更加健壮，在编写 Java 类时不妨养成这样的编程习惯：

- 如果 Java 类重新定义了 `equals()` 方法，那么这个类也必须重新定义 `hashCode()` 方法，并且保证当两个对象用 `equals()` 方法比较的结果为 `true` 时，这两个对象的 `hashCode()` 方法的返回值相等。
- 如果 Java 类实现了 `Comparable` 接口，那么应该重新定义 `compareTo()`、`equals()` 和 `hashCode()` 方法，保证 `compareTo()` 和 `equals()` 方法采用相同的比较规则来比较两个对象是否相等，并且保证当两个对象用 `equals()` 方法比较的结果为 `true` 时，这两个对象的 `hashCode()` 方法的返回值相等。

## 第 12 章 映射值类型集合

一个猴子可以拜多个武术师（假定武术师都是来自天届的各路神仙，非花果山的猴子）为师，一个武术师可以教多个猴子。因此 `Monkey` 类与 `Teacher` 类（代表武术师）之间为多对多关联关系，在 `Monkey` 类中定义了一个集合类型的属性 `teachers`，它用来存放所有与 `Monkey` 对象关联的 `Teacher` 对象。假如 `Monkey` 类还有一个集合类型的属性 `images`，用来存放 `Monkey` 对象的所有照片的文件名，那么 `images` 属性和 `teachers` 属性有相同的定义形式：

```
private Set teachers=new HashSet();
private Set images=new HashSet();
```

`teachers` 属性与 `images` 属性的区别在于，前者存放的是实体类型的 `Teacher` 对象，而后者存放的是值类型的 `String` 对象，本书第 8 章的 8.3.1 节（区分值类型和实体类型）介绍了实体类型和值类型的区别，实体类型的对象有单独的 `OID` 和独立的生命周期，而值类型的对象没有单独的 `OID` 和独立的生命周期。本章将介绍如何映射值类型的集合。

本书第 11 章（Java 集合）已经介绍过，按照集合的数据结构划分，Java 集合可分为 3 种：

- **Set**：集合中的对象不按特定方式排序，并且没有重复对象。它的有些实现类（如 `TreeSet`）能对集合中的对象按特定方式排序。
- **List**：集合中的对象按索引位置排序，可以有重复对象，允许按照对象在集合中的索引位置检索对象。
- **Map**：集合中的每一个元素包含一对键对象和值对象，集合中没有重复的键对象，值对象可以重复。它的有些实现类（如 `TreeMap`）能对集合中的键对象按特定方式排序。

Hibernate 允许把以上 3 种 Java 集合都映射到数据库中，在映射文件中，与映射 Java 集合相关的元素包括 `<set>`、`<idbag>`、`<list>` 和 `<map>`。

### 12.1 映射 Set（集）

假定 `Monkey` 对象的 `images` 集合中不允许存放重复的照片文件名，因此可以把 `images` 属性定义为 `Set` 类型：

```
private Set images=new HashSet();
public Set getImages() {
```

```

        return this.images;
    }
    public void setImages(Set images) {
        this.images = images;
    }

```

在数据库中定义了一张 IMAGES 表,它的 MONKEY\_ID 字段为参照 MONKEYS 表的外键,由于 Monkey 对象不允许有重复的照片文件名,因此应该把 IMAGES 表的 MONKEY\_ID 和 FILENAME 字段作为联合主键,图 12-1 显示了 MONKEYS 和 IMAGES 表的结构。

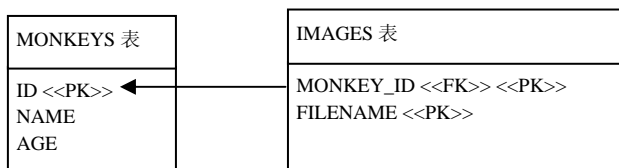


图 12-1 MONKEYS 表和 IMAGES 表的结构

以下是 IMAGES 表的 DDL 定义:

```

create table IMAGES(
    MONKEY_ID bigint not null,
    FILENAME varchar(15) not null,
    primary key (MONKEY_ID,FILENAME)
);
alter table IMAGES add index IDX_MONKEY(MONKEY_ID),
add constraint FK_MONKEY foreign key(MONKEY_ID) references MONKEYS(ID);

```

在 Monkey.hbm.xml 文件中,映射 Monkey 类的 images 属性的代码如下:

```

<set name="images" table="IMAGES" lazy="true">
    <key column="MONKEY_ID" />
    <element column="FILENAME" type="string" not-null="true"/>
</set>

```

<set>元素包含以下属性。

- name 属性: 指定 Monkey 类的 images 属性名。
- table 属性: 指定和 images 属性对应的表为 IMAGES 表。
- lazy 属性: 如果为 true, 表示对 images 集合使用延迟检索策略。由于 lazy 属性的默认值为 true, 所以此处也可以不显式设置 lazy 属性。

<set>元素的<key>子元素指定 IMAGES 表的外键为 MONKEY\_ID, <element>元素指定和 images 集合中元素对应的字段为 FILENAME 字段,并且 images 集合中的元素为字符串类型。

本节的范例程序位于配套光盘的 sourcecode\chapter12\12.1 目录下,运行该程序前,需要在 SAMPLEDB 数据库中手工创建 MONKEYS 和 IMAGES 表,相关的

SQL 脚本文件为 12.1\schema\sampledb.sql。

在 chapter12 目录下有 6 个 ANT 的工程文件，如 build1.xml、build2.xml、build3.xml 和 build4.xml 等，它们的区别在于文件开头设置的路径不一样，例如在 build1.xml 文件中设置了以下路径：

```
<property name="source.root" value="12.1/src"/>
<property name="class.root" value="12.1/classes"/>
<property name="lib.dir" value="lib"/>
<property name="schema.dir" value="12.1/schema"/>
```

在 DOS 命令行下进入 chapter12 根目录，然后输入命令：

```
ant -file build1.xml run
```

就会运行 BusinessService 类。ANT 命令的 -file 选项用于显式指定工程文件。例程 12-1 是 BusinessService 类的源程序。

例程 12-1 BusinessService.java

```
public class BusinessService{
    public static SessionFactory sessionFactory;
    /** 初始化 Hibernate, 创建 SessionFactory 对象
    static{.....}
    public void saveMonkey(Monkey monkey) {.....}
    public Monkey loadMonkey(long id) {.....}

    public void test(){
        Set images=new HashSet();
        images.add("image1.jpg");
        images.add("image4.jpg");
        images.add("image2.jpg");
        images.add("image5.jpg");

        Monkey monkey=new Monkey("Tom",21,images);
        saveMonkey(monkey);
        monkey=loadMonkey(1);
        printMonkey(monkey);
    }

    private void printMonkey(Monkey monkey){
        System.out.println(monkey.getImages().getClass().getName());
        Iterator it=monkey.getImages().iterator();
        while(it.hasNext()){
            String fileName=(String)it.next();
            System.out.println(monkey.getName()+" "+fileName);
        }
    }
}
```

```
public static void main(String args[]){
    new BusinessService().test();
    sessionFactory.close();
}
}
```

BusinessService 类的 main()方法调用 test()方法，test()方法依次调用以下方法。

- saveMonkey(): 保存一个 Monkey 对象。
- loadMonkey(): 加载一个 Monkey 对象。
- printMonkey(): 打印 Monkey 对象的信息，包括它的 images 集合中的所有照片文件名。

(1) 运行 saveMonkey (Monkey monkey)方法，它的代码如下：

```
tx = session.beginTransaction();
session.save(monkey);
tx.commit();
```

在 test()方法中创建了一个 Monkey 实例，然后调用 saveMonkey()方法保存这个实例：

```
Set images=new HashSet();
images.add("image1.jpg");
images.add("image4.jpg");
images.add("image2.jpg");
images.add("image5.jpg");

Monkey monkey=new Monkey("Tom",21,images);
saveMonkey(monkey);
```

Session 的 save()方法向 MONKEYS 表插入一条记录，同时还会向 IMAGES 表插入四条记录，执行如下 insert 语句：

```
insert into MONKEYS (ID,NAME, AGE) values (1, 'Tom', 21);
insert into IMAGES (MONKEY_ID, FILENAME) values (1, 'image1.jpg');
insert into IMAGES (MONKEY_ID, FILENAME) values (1, 'image4.jpg');
insert into IMAGES (MONKEY_ID, FILENAME) values (1, 'image2.jpg');
insert into IMAGES (MONKEY_ID, FILENAME) values (1, 'image5.jpg');
```

(2) 运行 loadMonkey()方法，它的代码如下：

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(id));
Hibernate.initialize(monkey.getImages());
tx.commit();
return monkey;
```

由于在 Monkey.hbm.xml 文件中对 images 集合使用了延迟检索策略，因此必须



通过 Hibernate 类的 `initialize()` 方法显示初始化 `images` 集合, 这样才能保证当 `Monkey` 对象成为游离对象后, `BusinessService` 类的 `test()` 方法能够正常访问 `images` 集合中的元素。Hibernate 类的 `initialize()` 方法执行以下 `select` 语句:

```
select MONKEY_ID,FILENAME from IMAGES where MONKEY_ID=1;
```

(3) 运行 `printMonkey()` 方法, 它的代码如下:

```
System.out.println(monkey.getImages().getClass().getName());
Iterator it=monkey.getImages().iterator();
while(it.hasNext()){
    String fileName=(String)it.next();
    System.out.println(monkey.getName()+" "+fileName);
}
```

以上程序的输出结果为:

```
org.hibernate.collection.PersistentSet
Tom image5.jpg
Tom image1.jpg
Tom image4.jpg
Tom image2.jpg
```

从以上输出结果看出, 当 Hibernate 加载 `Monkey` 对象的 `images` 集合时, 创建的是 `org.hibernate.collection.PersistentSet` 实例, `PersistentSet` 类实现了 `java.util.Set` 接口。此外, `Monkey` 对象的 `images` 集合中的元素不会保持固定顺序, 在 `test()` 方法中向 `Monkey` 对象的 `images` 集合加入元素的顺序为:

```
image1.jpg、image4.jpg、image2.jpg、image5.jpg
```

而 Hibernate 加载的 `Monkey` 对象的 `images` 集合中的元素的顺序为:

```
image5.jpg、image1.jpg、image4.jpg、image2.jpg
```

## 12.2 映射 Bag (包)

Bag 集合中的对象不按特定方式排序, 但是允许有重复对象。在 Java 集合 API 中并没有提供 Bag 接口, Hibernate 允许在持久化类中用 `List` 来模拟 Bag 的行为。假定 `Monkey` 对象的 `images` 集合中允许存放重复的照片文件名, 可以把 `images` 属性定义为 `List` 类型:

```
private List images=new ArrayList();
public List getImages() {
    return this.images;
}
public void setImages(List images) {
    this.images = images;
}
```

在数据库中定义了一张 IMAGES 表,它的 MONKEY\_ID 字段为参照 MONKEYS 表的外键,由于 Monkey 对象允许有重复的照片文件名,因此应该在 IMAGES 表中定义一个代理主键 ID,图 12-2 显示了 MONKEYS 和 IMAGES 表的结构。

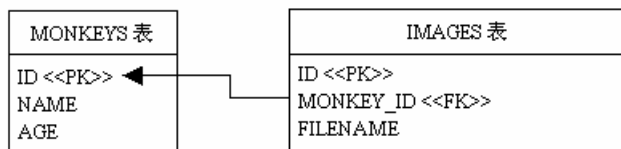


图 12-2 MONKEYS 表和 IMAGES 表的结构

以下是 IMAGES 表的 DDL 定义:

```

create table IMAGES(
    ID bigint not null,
    MONKEY_ID bigint not null,
    FILENAME varchar(15) not null,
    primary key (ID)
);
alter table IMAGES add index IDX_MONKEY(MONKEY_ID),
add constraint FK_MONKEY foreign key (MONKEY_ID) references MONKEYS(ID);
  
```

在 Monkey.hbm.xml 文件中,映射 Monkey 类的 images 属性的代码如下:

```

<idbag name="images" table="IMAGES" lazy="true">
    <collection-id type="long" column="ID">
        <generator class="increment"/>
    </collection-id>
    <key column="MONKEY_ID" />
    <element column="FILENAME" type="string" not-null="true"/>
</idbag>
  
```

<idbag>元素与本章 12.1 节介绍的<set>元素的配置很相似,区别在于<idbag>中增加了<collection-id>子元素,它用于设置 IMAGES 表的 ID 主键。

本节的范例程序位于配套光盘的 sourcecode\chapter12\12.2 目录下,运行该程序前,需要在 SAMPLEDB 数据库中手工创建 MONKEYS 和 IMAGES 表,相关的 SQL 脚本文件为 12.2\schema\sampldb.sql。

在 DOS 命令行下进入 chapter12 根目录,然后输入命令:

```
ant -file build2.xml run
```

就会运行 BusinessService 类。BusinessService 类的源程序和本章 12.1 节的例程 12-1 很相似。BusinessService 的 main()方法调用 test()方法, test()方法依次调用以下方法:

- `saveMonkey()`: 保存一个 `Monkey` 对象。
- `loadMonkey()`: 加载一个 `Monkey` 对象。
- `printMonkey()`: 打印 `Monkey` 对象的信息, 包括它的 `images` 集合中的所有图片文件名。

(1) 运行 `saveMonkey(Monkey monkey)` 方法。在 `test()` 方法中创建了一个 `Monkey` 实例, 然后调用 `saveMonkey()` 方法保存这个实例:

```
List images=new ArrayList();
images.add("image1.jpg");
images.add("image4.jpg");
images.add("image2.jpg");
images.add("image2.jpg");
images.add("image5.jpg");

Monkey monkey=new Monkey("Tom",21,images);
saveMonkey(monkey);
```

`Session` 的 `save()` 方法向 `MONKEYS` 表插入一条记录, 同时还会向 `IMAGES` 表插入 5 条记录, 执行如下 `insert` 语句:

```
insert into MONKEYS (ID,NAME, AGE) values (1, 'Tom', 21);
insert into IMAGES (ID,MONKEY_ID, FILENAME) values (1, 1,'image1.jpg');
insert into IMAGES (ID,MONKEY_ID, FILENAME) values (2, 1,'image4.jpg');
insert into IMAGES (ID,MONKEY_ID, FILENAME) values (3, 1,'image2.jpg');
insert into IMAGES (ID,MONKEY_ID, FILENAME) values (4, 1,'image2.jpg');
insert into IMAGES (ID,MONKEY_ID, FILENAME) values (5, 1,'image5.jpg');
```

(2) 运行 `loadMonkey()` 方法, 它的代码如下:

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(id));
Hibernate.initialize(monkey.getImages());
tx.commit();
return monkey;
```

由于在 `Monkey.hbm.xml` 文件中对 `images` 集合使用了延迟检索策略, 因此必须通过 `Hibernate` 类的 `initialize()` 方法显示初始化 `images` 集合, 这样才能保证当 `Monkey` 对象成为游离对象后, `BusinessService` 类的 `test()` 方法能够正常访问 `images` 集合中的元素。Hibernate 类的 `initialize()` 方法执行以下 `select` 语句:

```
select ID,MONKEY_ID,FILENAME from IMAGES where MONKEY_ID=1;
```

(3) 运行 `printMonkey()` 方法, 它的代码如下:

```
System.out.println(monkey.getImages().getClass().getName());
Iterator it=monkey.getImages().iterator();
while(it.hasNext()){
```

```
String fileName=(String)it.next();
System.out.println(monkey.getName()+" "+fileName);
}
```

以上程序的输出结果为：

```
org.hibernate.collection.PersistentIdentifierBag
Tom image1.jpg
Tom image4.jpg
Tom image2.jpg
Tom image2.jpg
Tom image5.jpg
```

从以上输出结果看出，当 Hibernate 加载 Monkey 对象的 images 集合时，创建的是 org.hibernate.collection.PersistentIdentifierBag 实例，PersistentIdentifierBag 类实现了 java.util.List 接口。此外，在 test()方法中向 Monkey 对象的 images 集合加入元素的顺序为：

```
image1.jpg、image4.jpg、image2.jpg、image2.jpg、image5.jpg
```

从以上程序的输出结果看出，Hibernate 加载的 Monkey 对象的 images 集合中的元素也采用相同的顺序。尽管这两者的顺序相同，但这只是偶然情况，事实上，Hibernate 不会保证 Bag 集合中的元素保持固定的顺序，因此在程序中应该避免通过以下方式访问 images 集合中的元素：

```
Monkey monkey=loadMonkey(1);
List images=monkey.getImages();
String fileName=(String)images.get(4);
```

以上程序按照索引位置检索 images 集合中的元素，但是由于 Hibernate 并不会保证每个元素有固定的索引位置，因此多次执行该程序时，images.get(4)方法有可能返回 Bag 集合中的不同元素。

## 12.3 映射List（列表）

在本章 12.2 节中，尽管 Monkey 类的 images 属性被定义为 List 类型，但是由于在 Monkey.hbm.xml 文件中用<idbag>元素来映射它，因此 images 集合中的元素并不会按照索引位置排序。如果希望 images 集合中允许存放重复元素，并且按照索引位置排序，首先应该在 IMAGES 表中定义一个 POSITION 字段，代表每个元素在集合中的索引位置。MONKEY\_ID 和 POSITION 字段共同构成了 IMAGES 表的主键，图 12-3 显示了 MONKEYS 和 IMAGES 表的结构。

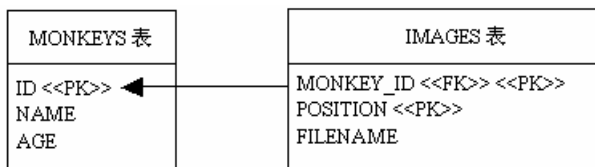


图 12-3 MONKEYS 表和 IMAGES 表的结构

以下是 IMAGES 表的 DDL 定义：

```

create table IMAGES(
    MONKEY_ID bigint not null,
    POSITION int not null,
    FILENAME varchar(15) not null,
    primary key (MONKEY_ID, POSITION)
);

alter table IMAGES add index IDX_MONKEY(MONKEY_ID),
add constraint FK_MONKEY foreign key (MONKEY_ID) references MONKEYS(ID);
  
```

在 Monkey.hbm.xml 文件中，映射 Monkey 类的 images 属性的代码如下：

```

<list name="images" table="IMAGES" lazy="true">
    <key column="MONKEY_ID" />
    <list-index column="POSITION" />
    <element column="FILENAME" type="string" not-null="true"/>
</list>
  
```

<list>元素与 12.1 节介绍的<set>元素的配置很相似，区别在于<list>中增加了<list-index>子元素，它用于设置 IMAGES 表中代表索引位置的 POSITION 字段。

本节的范例程序位于配套光盘的 sourcecode\chapter12\12.3 目录下，运行该程序前，需要在 SAMPLEDB 数据库中手工创建 MONKEYS 和 IMAGES 表，相关的 SQL 脚本文件为 12.3\schema\sampldb.sql。

在 DOS 命令行下进入 chapter12 根目录，然后输入命令：

```
ant -file build3.xml run
```

就会运行 BusinessService 类。BusinessService 类的源程序和本章 12.1 节的例程 12-1 很相似。BusinessService 的 main()方法调用 test()方法，test()方法依次调用以下方法：

- saveMonkey(): 保存一个 Monkey 对象。
- loadMonkey(): 加载一个 Monkey 对象。
- printMonkey(): 打印 Monkey 对象的信息，包括它的 images 集合中的所有图片文件名。

(1) 运行 `saveMonkey (Monkey monkey)`方法。在 `test()`方法中创建了一个 `Monkey` 实例，然后调用 `saveMonkey()`方法保存这个实例：

```
List images=new ArrayList();
images.add("image1.jpg");
images.add("image4.jpg");
images.add("image2.jpg");
images.add("image2.jpg");
images.add("image5.jpg");

Monkey monkey=new Monkey("Tom",21,images);
saveMonkey(monkey);
```

`Session` 的 `save()`方法向 `MONKEYS` 表插入一条记录，同时还会向 `IMAGES` 表插入 5 条记录，执行如下 `insert` 语句：

```
insert into MONKEYS (ID,NAME, AGE) values (1, 'Tom', 21);
insert into IMAGES (POSITION,MONKEY_ID, FILENAME) values (0,1,
    'image1.jpg');
insert into IMAGES (POSITION,MONKEY_ID, FILENAME) values (1,
    1,'image4.jpg');
insert into IMAGES (POSITION,MONKEY_ID, FILENAME) values (2,
    1,'image2.jpg');
insert into IMAGES (POSITION,MONKEY_ID, FILENAME) values (3,
    1,'image2.jpg');
insert into IMAGES (POSITION,MONKEY_ID, FILENAME) values (4,
    1,'image5.jpg');
```

`Monkey` 对象的 `images` 集合中的第一个元素的索引位置为 0，第二次元素的索引位置为 1，依次类推。假如应用程序向数据库保存第二个 `Monkey` 对象：

```
List images=new ArrayList();
images.add("file2.jpg");
images.add("file1.jpg");

Monkey monkey=new Monkey("Mike",25,images);
saveMonkey(monkey);
```

那么 `Monkey` 对象的 `images` 集合中的元素的索引位置仍然从 0 开始计数，`Session` 的 `save()`方法执行如下 `insert` 语句：

```
insert into MONKEYS (ID,NAME, AGE) values (2, 'Mike', 25);
insert into IMAGES (POSITION,MONKEY_ID, FILENAME) values (0,2,
    'file2.jpg');
insert into IMAGES (POSITION,MONKEY_ID, FILENAME) values (1,
    2,'file1.jpg');
```

(2) 运行 `loadMonkey()`方法，它的代码如下所示：

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(id));
Hibernate.initialize(monkey.getImages());
tx.commit();
return monkey;
```

由于在 Monkey.hbm.xml 文件中对 images 集合使用了延迟检索策略, 因此必须通过 Hibernate 类的 initialize() 方法显示初始化 images 集合, 这样才能保证当 Monkey 对象成为游离对象后, BusinessService 类的 test() 方法能够正常访问 images 集合中的元素。Hibernate 类的 initialize() 方法执行以下 select 语句:

```
select POSITION,MONKEY_ID,FILENAME from IMAGES where MONKEY_ID=1;
```

(3) 运行 printMonkey() 方法, 它的代码如下所示:

```
System.out.println(monkey.getImages().getClass().getName());
List images=monkey.getImages();
for(int i=images.size()-1;i>=0;i--){
    String fileName=(String)images.get(i);
    System.out.println(monkey.getName()+" "+fileName);
}
```

以上程序的输出结果为:

```
org.hibernate.collection.PersistentList
Tom image5.jpg
Tom image2.jpg
Tom image2.jpg
Tom image4.jpg
Tom image1.jpg
```

从以上输出结果看出, 当 Hibernate 加载 Monkey 对象的 images 集合时, 创建的是 org.hibernate.collection.PersistentList 实例, PersistentList 类实现了 java.util.List 接口。由于 Monkey.hbm.xml 文件用 <list> 元素来映射 Monkey 类的 images 属性, Hibernate 会保证 images 集合中的每个元素有固定的索引位置, 因此在程序中可以通过 images.get(i) 方法来检索 images 集合中的元素。

## 12.4 映射 Map

如果 Monkey 类的 images 集合中的每一个元素包含一对键对象和值对象, 那么应该把 images 集合定义为 Map 类型:

```
private Map images=new HashMap();
public Map getImages() {
    return this.images;
}
```

```
public void setImages(Map images) {
    this.images = images;
}
```

在 IMAGES 表中定义了一个 IMAGE\_NAME 字段，它和 images 集合中的键对象对应，表示图片名字。MONKEY\_ID 和 IMAGE\_NAME 字段共同构成了 IMAGES 表的主键，图 12-4 显示了 MONKEYS 和 IMAGES 表的结构。

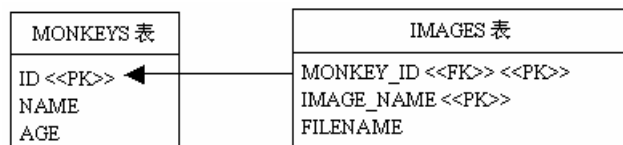


图 12-4 MONKEYS 表和 IMAGES 表的结构

表 12-1 显示了 IMAGES 表中的数据。

表 12-1 IMAGES 表中的数据		
MONKEY_ID	IMAGE_NAME	FILENAME
1	image1	image1.jpg
1	image4	image4.jpg
1	image2	image2.jpg
1	imageTwo	image2.jpg
1	image5	image5.jpg
2	file1	file1.jpg
2	file2	file2.jpg

以下是 IMAGES 表的 DDL 定义：

```
create table IMAGES(
    MONKEY_ID bigint not null,
    IMAGE_NAME varchar(15) not null,
    FILENAME varchar(15) not null,
    primary key (MONKEY_ID,IMAGE_NAME)
);
alter table IMAGES add index IDX_MONKEY(MONKEY_ID),
add constraint FK_MONKEY foreign key (MONKEY_ID) references
    MONKEYS(ID);
```

在 Monkey.hbm.xml 文件中，映射 Monkey 类的 images 属性的代码如下所示：

```
<map name="images" table="IMAGES" lazy="true">
    <key column="MONKEY_ID" />
    <map-key column="IMAGE_NAME" type="string"/>
    <element column="FILENAME" type="string" not-null="true"/>
</map>
```



<map>元素与 12.1 节介绍的<set>元素的配置很相似,区别在于<map>元素中增加了<map-key>子元素,它用于设置 IMAGES 表中和 images 集合的键对象对应的 IMAGE\_NAME 字段。

本节的范例程序位于配套光盘的 sourcecode\chapter12\12.4 目录下,运行该程序前,需要在 SAMPLEDB 数据库中手工创建 MONKEYS 和 IMAGES 表,相关的 SQL 脚本文件为 12.4\schema\sampladb.sql。

在 DOS 命令行下进入 chapter12 根目录,然后输入命令:

```
ant -file build4.xml run
```

就会运行 BusinessService 类。BusinessService 类的源程序和本章 12.1 节的例程 12-1 很相似。BusinessService 的 main()方法调用 test()方法, test()方法依次调用以下方法:

- saveMonkey(): 保存一个 Monkey 对象。
- loadMonkey(): 加载一个 Monkey 对象。
- printMonkey(): 打印 Monkey 对象的信息,包括它的 images 集合中的所有图片文件名。

(1) 运行 saveMonkey (Monkey monkey)方法。在 test()方法中创建了一个 Monkey 实例,然后调用 saveMonkey()方法保存这个实例:

```
Map images=new HashMap();
images.put("image1","image1.jpg");
images.put("image4","image4.jpg");
images.put("image2","image2.jpg");
images.put("imageTwo","image2.jpg");
images.put("image5","image5.jpg");

Monkey monkey=new Monkey("Tom",21,images);
saveMonkey(monkey);
```

Session 的 save()方法向 MONKEYS 表插入一条记录,同时还会向 IMAGES 表插入 5 条记录,执行如下 insert 语句:

```
insert into MONKEYS(ID,NAME, AGE) values (1, 'Tom', 21);
insert into IMAGES(IMAGE_NAME,MONKEY_ID,FILENAME)values('image1',1,
'image1.jpg');
insert into IMAGES(IMAGE_NAME,MONKEY_ID,FILENAME)values('image4',1,
'image4.jpg');
insert into IMAGES(IMAGE_NAME,MONKEY_ID,FILENAME)values('image2',1,
'image2.jpg');
insert into IMAGES(IMAGE_NAME,MONKEY_ID,FILENAME)
values('imageTwo',1,'image2.jpg');
insert into IMAGES(IMAGE_NAME,MONKEY_ID,FILENAME)values('image5',1,
'image5.jpg');
```

(2) 运行 loadMonkey()方法，它的代码如下所示：

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(id));
Hibernate.initialize(monkey.getImages());
tx.commit();
return monkey;
```

由于在 Monkey.hbm.xml 文件中对 images 集合使用了延迟检索策略，因此必须通过 Hibernate 类的 initialize()方法显示初始化 images 集合，这样才能保证当 Monkey 对象成为游离对象后，BusinessService 类的 test()方法能够正常访问 images 集合中的元素。Hibernate 类的 initialize()方法执行以下 select 语句：

```
select IMAGE_NAME,MONKEY_ID,FILENAME from IMAGES where MONKEY_ID=1;
```

(3) 运行 printMonkey()方法，它的代码如下所示：

```
System.out.println(monkey.getImages().getClass().getName());
Map images=monkey.getImages();
Set keys=images.keySet();
Iterator it=keys.iterator();
while(it.hasNext()){
    String key=(String)it.next();
    String filename=(String)images.get(key);
    System.out.println(monkey.getName()+" "+key+" "+filename);
}
```

以上程序的输出结果为：

```
org.hibernate.collection.PersistentMap
Tom imageTwo image2.jpg
Tom image2 image2.jpg
Tom image4 image4.jpg
Tom image5 image5.jpg
Tom image1 image1.jpg
```

从以上输出结果看出，当 Hibernate 加载 Monkey 对象的 images 集合时，创建的是 org.hibernate.collection.PersistentMap 实例，PersistentMap 类实现了 java.util.Map 接口。此外，Hibernate 不会对 images 集合中的键对象进行排序。

## 12.5 对集合排序

Hibernate 对集合中的元素支持两种排序方式：

- 在数据库中排序：简称为数据库排序，当 Hibernate 通过 select 语句到数据库中检索集合对象时，利用 order by 子句进行排序。

- 在内存中排序：简称为内存排序，当 Hibernate 把数据库中的集合数据加载到内存中的 Java 集合中后，利用 Java 集合的排序功能进行排序，可以选择自然排序或者客户化排序两种方式。本书第 11 章介绍了 Java 集合的自然排序和客户化排序方式。

在映射文件中，Hibernate 用 `sort` 属性来设置内存排序，用 `order-by` 属性来设置数据库排序，表 12-2 显示了 `<set>`、`<idbag>`、`<list>` 和 `<map>` 元素的排序属性。

表 12-2 `<set>`、`<idbag>`、`<list>` 和 `<map>` 元素的排序属性

排 序 属 性	<code>&lt;set&gt;</code>	<code>&lt;idbag&gt;</code>	<code>&lt;list&gt;</code>	<code>&lt;map&gt;</code>
<code>sort</code> 属性（内存排序）	支持	不支持	不支持	支持
<code>order-by</code> 属性（数据库排序）	支持	支持	不支持	支持

从表 12-2 看出，`<set>` 和 `<map>` 元素支持内存排序和数据库排序，`<list>` 元素不支持任何排序方式，而 `<idbag>` 仅支持数据库排序。

### 12.5.1 在数据库中对集合排序

`<set>`、`<idbag>` 和 `<map>` 元素都具有 `order-by` 属性，如果设置了该属性，当 Hibernate 通过 `select` 语句到数据库中检索集合对象时，利用 `order by` 子句进行排序。

下面对本章 12.1 节的 `Monkey.hbm.xml` 文件中的 `<set>` 元素增加一个 `order-by` 属性：

```
<set name="images" table="IMAGES" lazy="true" order-by=
    "FILENAME asc">
    <key column="MONKEY_ID" />
    <element column="FILENAME" type="string" not-null="true"/>
</set>
```

以上代码表明对 `images` 集合中的元素进行升序排列，当 Hibernate 加载 Monkey 对象的 `images` 集合时，执行的 `select` 语句为：

```
select MONKEY_ID,FILENAME from IMAGES
where MONKEY_ID=1 order by FILENAME;
```

在 DOS 命令行下进入 `chapter12` 根目录，然后输入命令：

```
ant -file build1.xml run
```

就会运行 `BusinessService` 类。`BusinessService` 的 `main()` 方法调用 `test()` 方法，它的输出结果如下：

```
org.hibernate.collection.PersistentSet
Tom image1.jpg
Tom image2.jpg
Tom image4.jpg
Tom image5.jpg
```

在 `order-by` 属性中还可以加入 SQL 函数，例如：

```
<set name="images" table="IMAGES" lazy="true"
    order-by="lower(FILENAME) desc">

    <key column="MONKEY_ID" />
    <element column="FILENAME" type="string" not-null="true"/>
</set>
```

当 Hibernate 加载 Monkey 对象的 `images` 集合时，执行的 `select` 语句为：

```
select MONKEY_ID,FILENAME from IMAGES
where MONKEY_ID=1 order by lower(FILENAME) desc;
```

在 `<map>` 元素中也可以加入 `order-by` 属性，以下代码表明对 Map 类型的 `images` 集合中的键对象进行排序：

```
<map name="images" table="IMAGES" lazy="true" order-by=
    "IMAGE_NAME">
    <key column="MONKEY_ID" />
    <map-key column="IMAGE_NAME" type="string"/>
    <element column="FILENAME" type="string" not-null="true"/>
</map>
```

以下代码表明对 Map 类型的 `images` 集合中的值对象进行排序：

```
<map name="images" table="IMAGES" lazy="true" order-by=
    "FILENAME">
    <key column="MONKEY_ID" />
    <map-key column="IMAGE_NAME" type="string"/>
    <element column="FILENAME" type="string" not-null="true"/>
</map>
```

在 `<idbag>` 元素中也可以加入 `order-by` 属性，以下代码表明按照 `IMAGES` 表中的 ID 代理主键排序：

```
<idbag name="images" table="IMAGES" lazy="true"
    order-by="ID">
    <collection-id type="long" column="ID">
        <generator class="increment"/>
    </collection-id>
    <key column="MONKEY_ID" />
    <element column="FILENAME" type="string" not-null="true"/>
</idbag>
```

## 12.5.2 在内存中对集合排序

`<set>` 和 `<map>` 元素都具有 `sort` 属性，如果设置了该属性，就会对内存中的集合对象进行排序。

## 1. <set>元素在内存中对集合排序

下面对本章 12.1 节的 Monkey.hbm.xml 文件中的<set>元素增加一个 sort 属性：

```
<set name="images" table="IMAGES" lazy="true" sort="natural">
    <key column="MONKEY_ID" />
    <element column="FILENAME" type="string" not-null="true"/>
</set>
```

<set>元素的 sort 属性为 natural, 表示对 images 集合中的字符串进行自然排序。Hibernate 采用 org.hibernate.PersistentSortedSet 作为 Set 的实现类, PersistentSortedSet 类实现了 java.util.SortedSet 接口。当 Session 保存一个 Monkey 对象时, 会调用 org.hibernate.type.SortedSetType 类的 wrap() 方法, 把 Monkey 对象的 images 集合包装为 SortedSet 类的实例, wrap() 方法的代码如下：

```
public PersistentCollection wrap(SessionImplementor session,
    Object collection) {
    return new PersistentSortedSet(session, (java.util.SortedSet)collection);
}
```

从 wrap() 方法的源代码看出, 应用程序中创建的 Monkey 对象的 images 集合必须是 java.util.SortedSet 类型, 否则以上 wrap() 方法会抛出 ClassCastException。由于 java.util.TreeSet 类实现了 java.util.SortedSet 接口, 因此在 Monkey 类中初始化 images 属性时, 可以创建 TreeSet 类型的实例：

```
private Set images=new TreeSet();
public Set getImages() {
    return this.images;
}
public void setImages(Set images) {
    this.images = images;
}
```

在 BusinessService 类的 test() 方法中也应该创建 TreeSet 类的实例：

```
Set images=new TreeSet();
images.add("image1.jpg");
images.add("image4.jpg");
images.add("image2.jpg");
images.add("image5.jpg");

Monkey monkey=new Monkey("Tom", 21, images);
saveMonkey(monkey);

monkey=loadMonkey(1);
printMonkey(monkey);
```

本节的范例程序位于配套光盘的 sourcecode\chapter12\12.5.1 目录下, 运行该程

序前，需要在 SAMPLEDB 数据库中手工创建 MONKEYS 表和 IMAGES 表，相关的 SQL 脚本文件为\12.5.1\schema\sampledb.sql。在 DOS 命令行下进入 chapter12 根目录，然后输入命令：

```
ant -file build5.1.xml run
```

就会运行 BusinessService 类。BusinessService 类的源程序和本章 12.1 节的例程 12-1 很相似。BusinessService 的 main()方法调用 test()方法，它的输出结果如下：

```
org.hibernate.collection.PersistentSortedSet
Tom image1.jpg
Tom image2.jpg
Tom image4.jpg
Tom image5.jpg
```

从输出结果看出，当 Hibernate 加载 Monkey 对象的 images 集合时，创建的是 org.hibernate.collection.PersistentSortedSet 实例，PersistentSortedSet 类实现了 java.util.SortedSet 接口，具有排序功能。

<set>元素也支持客户化排序。例程 12-2 的 ReverseStringComparator 类定义了一种对字符串进行降序排列的排序方式。

**例程 12-2 ReverseStringComparator.java**

```
public class ReverseStringComparator implements Comparator{
    public int compare(Object o1,Object o2){
        String s1=(String)o1;
        String s2=(String)o2;

        if(s1.compareTo(s2)>0) return -1;
        if(s1.compareTo(s2)<0) return 1;

        return 0;
    }
}
```

接下来把<set>元素的 sort 属性设为“mypack.ReverseStringComparator”：

```
<set name="images" table="IMAGES" lazy="true"
    sort="mypack.ReverseStringComparator">

    <key column="MONKEY_ID" />
    <element column="FILENAME" type="string" not-null="true"/>
</set>
```

再次运行 BusinessService 类，最后的输出结果为：

```
org.hibernate.collection.PersistentSortedSet
Tom image5.jpg
Tom image4.jpg
```

```
Tom image2.jpg
Tom image1.jpg
```

可见，Hibernate 能够根据 `ReverseStringComparator` 类定义的客户化排序方式，对 `images` 集合中的字符串做降序排列。

## 2. <map>元素在内存中对集合排序

<map>元素允许在内存中对集合中的键对象进行排序：

```
<map name="images" table="IMAGES" lazy="true" sort="natural">
  <key column="MONKEY_ID" />
  <map-key column="IMAGE_NAME" type="string"/>
  <element column="FILENAME" type="string" not-null="true"/>
</map>
```

以上代码表明对 `images` 集合中的键对象进行自然排序。如果把<map>元素的 `sort` 属性设为 `mypack.ReverseStringComparator`，则表明采用 `ReverseStringComparator` 类定义的客户化排序方式，对 `images` 集合中的字符串类型的键对象进行降序排列：

```
<map name="images" table="IMAGES" lazy="true"
  sort="mypack.ReverseStringComparator ">

  <key column="MONKEY_ID" />
  <map-key column="IMAGE_NAME" type="string"/>
  <element column="FILENAME" type="string" not-null="true"/>
</map>
```

Hibernate 采用 `org.hibernate.PersistentSortedMap` 作为 `Map` 的实现类，`PersistentSortedMap` 类实现了 `java.util.SortedMap` 接口。当 Session 保存一个 `Monkey` 对象时，会调用 `org.hibernate.type.SortedMapType` 类的 `wrap()` 方法，把 `Monkey` 对象的 `images` 集合包装为 `PersistentSortedMap` 类的实例，`wrap()` 方法的代码如下：

```
public PersistentCollection wrap(SessionImplementor session,
    Object collection){
    return new PersistentSortedMap(session, (java.util.SortedMap)
        collection );
}
```

从 `wrap()` 方法的源代码看出，应用程序中创建的 `Monkey` 对象的 `images` 集合必须是 `java.util.SortedMap` 类型，否则以上 `wrap()` 方法会抛出 `ClassCastException`。由于 `java.util.TreeMap` 类实现了 `java.util.SortedMap` 接口，因此在 `Monkey` 类中初始化 `images` 属性时，可以创建 `TreeMap` 类型的实例：

```
private Map images=new TreeMap();
public Map getImages() {
    return this.images;
}
```

```
public void setImages(Map images) {
    this.images = images;
}
```

在 `BusinessService` 类的 `test()` 方法中，也应该创建 `TreeMap` 类型的实例：

```
Map images=new TreeMap();
images.put("image1","image1.jpg");
images.put("image4","image4.jpg");
images.put("image2","image2.jpg");
images.put("imageTwo","image2.jpg");
images.put("image5","image5.jpg");

Monkey monkey=new Monkey("Tom",21,images);
saveMonkey(monkey);

monkey=loadMonkey(1);
printMonkey(monkey);
```

本节的范例程序位于配套光盘的 `sourcecode\chapter12\12.5.2` 目录下，运行该程序前，需要在 `SAMPLEDB` 数据库中手工创建 `MONKEYS` 表和 `IMAGES` 表，相关的 SQL 脚本文件为 `12.5.2\schema\sampladb.sql`。在 DOS 命令行下进入 `chapter12` 根目录，然后输入命令：

```
ant -file build5.2.xml run
```

就会运行 `BusinessService` 类。`BusinessService` 类的源程序和本章 12.1 节的例程 12-1 很相似。`BusinessService` 的 `main()` 方法调用 `test()` 方法，当 `Monkey.hbm.xml` 文件中 `<map>` 元素的 `sort` 属性为 `natural` 时，`test()` 方法的输出结果如下：

```
org.hibernate.collection.PersistentSortedMap
Tom image1 image1.jpg
Tom image2 image2.jpg
Tom image4 image4.jpg
Tom image5 image5.jpg
Tom imageTwo image2.jpg
```

从输出结果看出，当 Hibernate 加载 `Monkey` 对象的 `images` 集合时，创建的是 `org.hibernate.collection.PersistentSortedMap` 实例，`PersistentSortedMap` 类实现了 `java.util.SortedMap` 接口，具有排序功能。

## 12.6 小结

本章介绍了值类型集合的映射方法，在这种集合中存放的对象没有 `OID`，它们的生命周期依赖于集合所属的对象的生命周期。Hibernate 采用 `<set>`、`<list>` 和 `<map>`



元素来映射 `java.util.Set`、`java.util.List` 和 `java.util.Map`，此外 Hibernate 使用 `<idbag>` 元素来映射 Bag 集合，Bag 集合中的元素允许重复，但是不按特定方式排序，在 Java 类中没有提供 Bag 接口，Hibernate 允许在持久化类中用 `java.util.List` 来模拟 Bag 的行为。

对于每一种 Java 集合接口，Hibernate 都提供了内置的实现类，包括：

- `org.hibernate.collection.PersistentSet`
- `org.hibernate.collection.PersistentSortedSet`
- `org.hibernate.collection.PersistentIdentifierBag`
- `org.hibernate.collection.PersistentList`
- `org.hibernate.collection.PersistentMap`
- `org.hibernate.collection.PersistentSortedMap`

当 Session 从数据库中加载 Java 集合时，会创建以上内置集合类的实例。本书第 11 章已经介绍过，在 JDK 中也提供了现成的 Java 集合实现类，如 `java.util.HashSet`、`java.util.TreeSet`、`java.util.ArrayList`、`java.util.HashMap` 和 `java.util.TreeMap` 等。那么，Hibernate 为什么不直接使用 JDK 中现成的 Java 集合实现类呢？主要有以下原因：

- Hibernate 的内置集合类具有集合代理功能，支持延迟检索策略。如果对集合使用了延迟检索策略，只有当初始化集合时，才会真正加载集合中的对象。
- JDK 中没有 Bag 接口，Hibernate 的内置 `PersistentIdentifierBag` 类能够模拟 Bag 集合的行为。
- 事实上，Hibernate 的内置集合类都封装了 JDK 中的集合类，如 `PersistentSet` 封装了 `java.util.HashSet` 类，`PersistentSortedSet` 封装了 `java.util.TreeSet` 类。Hibernate 的内置集合类对 JDK 中的集合类进行了包装，使得 Hibernate 能够对缓存中的集合对象进行脏检查，按照集合对象的状态变化来同步更新数据库。

由于当 Session 从数据库中加载 Java 集合时，创建的是 Hibernate 内置集合类的实例，因此在持久化类中定义集合属性时，必须把它定义为 Java 接口类型，如：

```
private Set images=new HashSet();
```

如果把以上 images 集合定义为 HashSet 类型：

```
private HashSet images=new HashSet();
```

那么当 Session 从数据库中加载 images 集合时，会把 `org.hibernate.collection.PersistentSet` 实例赋值给 images 属性，从而抛出 `ClassCastException` 异常。

## 第 13 章 映射实体关联关系

本书第 5 章介绍了映射一对多关联关系的方法，这是对象模型中最常见的关联关系。本章将介绍另外两种关联关系的映射：一对一关联和多对多关联。

本章以 `Monkey` 与 `Address` 类的关系为例，介绍映射一对一关联的各种方法，然后以 `Monkey` 与 `Teacher` 类的关系为例，介绍映射多对多关联的各种方法。

### 13.1 映射一对一关联

本书第 8 章(映射组成关系)介绍了 `Monkey` 类与 `Address` 类的组成关系，`Address` 类是组件类，它没有 `OID`，在数据库中没有对应的表，`Address` 对象的生命周期依赖于 `Monkey` 对象的生命周期。在 `Monkey` 类中定义了两个值类型的 `homeAddress` 和 `comAddress` 属性：

```
private Address homeAddress;  
private Address comAddress;
```

如果是从头设计对象模型和数据模型，应该优先考虑把 `Monkey` 类与 `Address` 类设计为组成关系。假如在数据库中已经存在独立的 `ADDRESSES` 表，并且 `Address` 类已经被设计为实体类，有单独的 `OID`，那么 `Monkey` 类与 `Address` 类之间就变成了一对一关联关系，图 13-1 为这两个类的类框图。`Hibernate` 提供了两种映射一对一关联关系的方法。

- 按照外键映射：在 `MONKEYS` 表中定义两个外键 `HOME_ADDRESS_ID` 和 `COM_ADDRESS_ID`，它们都参照 `ADDRESSES` 表的主键。
- 按照主键映射：`ADDRESSES` 表的 `ID` 字段既是主键，同时作为外键参照 `MONKEYS` 表的主键，也就是说，`ADDRESSES` 表与 `MONKEYS` 表共享主键。

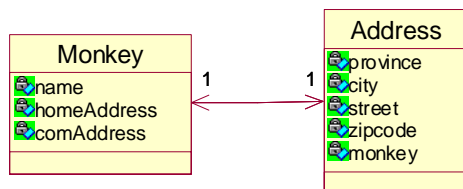


图 13-1 `Monkey` 类与 `Address` 类的一对一关联关系

### 13.1.1 按照外键映射

在图 13-2 中，MONKEYS 表的两个外键 HOME\_ADDRESS\_ID 和 COM\_ADDRESS\_ID 都参照 ADDRESSES 表的主键。

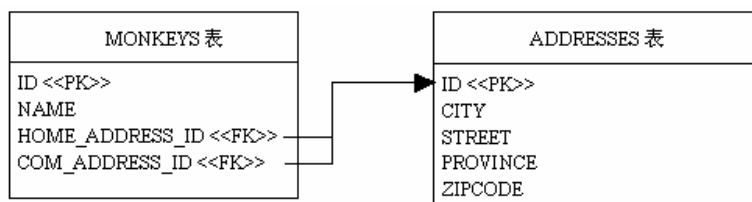


图 13-2 MONKEYS 表和 ADDRESSES 表的结构

以下是 MONKEYS 表的 DDL 定义：

```

create table MONKEYS (
    ID bigint not null,
    NAME varchar(15),
    HOME_ADDRESS_ID bigint unique,
    COM_ADDRESS_ID bigint unique,
    primary key (ID)
);

alter table MONKEYS add index IDX_HOME_ADDRESS(HOME_ADDRESS_ID),
add constraint FK_HOME_ADDRESS foreign key (HOME_ADDRESS_ID)
references ADDRESSES(ID);

alter table MONKEYS add index IDX_COM_ADDRESS(COM_ADDRESS_ID),
add constraint FK_COM_ADDRESS foreign key (COM_ADDRESS_ID)
references ADDRESSES(ID);
  
```

以上 MONKEYS 表的 HOME\_ADDRESS\_ID 和 COM\_ADDRESS\_ID 外键都设定了 unique 约束，确保每条 MONKEYS 记录都具有唯一的 HOME\_ADDRESS\_ID 和 COM\_ADDRESS\_ID。

在 Monkey.hbm.xml 文件中，用 <many-to-one> 元素来映射 Monkey 类的 homeAddress 和 comAddress 属性：

```

<many-to-one name="homeAddress"
    class="mypack.Address"
    column="HOME_ADDRESS_ID"
    cascade="all"
    unique="true"
/>

<many-to-one name="comAddress"
  
```

```

class="mypack.Address"
column="COM_ADDRESS_ID"
cascade="all"
unique="true"
/>

```

以上<many-to-one>元素的 cascade 属性为 all,表明当保存、更新或删除 Monkey 对象时,会级联保存、更新或删除 homeAddress 和 comAddress 对象。此外,<many-to-one>元素的 unique 属性为 true,表明每个 Monkey 对象都有唯一的 homeAddress 和 comAddress 对象。unique 属性的默认值为 false,如果把它设为 true,可以表达 Monkey 对象与 homeAddress 对象,以及 Monkey 对象与 comAddress 对象之间的一对一关联关系。

在 Address.hbm.xml 文件中,用<one-to-one>元素来映射 Address 类的 monkey 属性:

```

<one-to-one name="monkey"
    class="mypack.Monkey"
    property-ref="homeAddress"
/>

```

<one-to-one>元素的 property-ref 属性为 homeAddress,表明建立了从 homeAddress 对象到 Monkey 对象的关联。因此只要调用 homeAddress 持久化对象的 getMonkey()方法,就能导航到 Monkey 对象。以下程序代码从 Monkey 持久化对象导航到 homeAddress 持久化对象,又从 homeAddress 持久化对象导航到 Monkey 持久化对象,由此可见 Monkey 与 homeAddress 对象之间为双向关联关系:

```
monkey.getHomeAddress().getMonkey();
```

值得注意的是,在 Address.hbm.xml 文件中只能用<one-to-one>元素对 Address 类的 monkey 属性映射一次,因此这种映射方式只能映射 Monkey 对象与 homeAddress 对象的双向关联,但是不能同时映射 Monkey 对象与 comAddress 对象的双向关联。

#### Tips

如果希望同时映射 Monkey 对象与 homeAddress 对象,以及 Monkey 对象与 comAddress 对象的双向关联,一种解决办法是把 Address 类定义为抽象类,然后创建 HomeAddress 和 ComAddress 子类,在 HomeAddress.hbm.xml 和 ComAddress.hbm.xml 文件中分别用<one-to-one>元素来映射各自的 monkey 属性。这种解决办法的不足之处在于使对象模型变得更加复杂,因此应该谨慎使用这种方法。

本节的范例程序位于配套光盘的 sourcecode\chapter13\13.1.1 目录下,运行该程序前,需要在 SAMPLEDB 数据库中手工创建 MONKEYS 和 ADDRESSES 表,相关的 SQL 脚本文件为\13.1\schema\sampladb.sql。

在 chapter13 目录下有 6 个 ANT 的工程文件，如 build1.1.xml、build1.2.xml、build2.xml 和 build3.1.xml 等，它们的区别在于文件开头设置的路径不一样，例如在 build1.1.xml 文件中设置了以下路径：

```
<property name="source.root" value="13.1.1/src"/>
<property name="class.root" value="13.1.1/classes"/>
<property name="lib.dir" value="lib"/>
<property name="schema.dir" value="13.1.1/schema"/>
```

在 DOS 命令行下进入 chapter13 根目录，然后输入命令：

```
ant -file build1.1.xml run
```

就会运行 `BusinessService` 类。ANT 命令的 `-file` 选项用于显式指定工程文件。例程 13-1 是 `BusinessService` 类的源程序。

例程 13-1 `BusinessService.java`

```
public class BusinessService{
    public static SessionFactory sessionFactory;
    /** 初始化 Hibernate, 创建 SessionFactory 对象
    static{.....}
    public void saveMonkey(Monkey monkey) {.....}
    public Monkey loadMonkey(Long id) {.....}
    public void printMonkey(Monkey monkey) {.....}

    public void test(){
        Monkey monkey=new Monkey();
        Address homeAddress=
            new Address("province1","city1","street1","100001",monkey);
        Address comAddress=
            new Address("province2","city2","street2","200002",monkey);
        monkey.setName("Tom");
        monkey.setHomeAddress(homeAddress);
        monkey.setComAddress(comAddress);

        saveMonkey(monkey);
        monkey=loadMonkey(monkey.getId());
        printMonkey(monkey);
    }

    public static void main(String args[]){
        new BusinessService().test();
        sessionFactory.close();
    }
}
```

`BusinessService` 的 `main()` 方法调用 `test()` 方法，`test()` 方法依次调用以下方法。

- `saveMonkey()`: 保存一个 `Monkey` 对象。
- `loadMonkey()`: 加载一个 `Monkey` 对象。
- `printMonkey()`: 打印 `Monkey` 对象的信息, 包括它的 `homeAddress` 和 `comAddress` 信息。

(1) 运行 `saveMonkey (Monkey monkey)` 方法, 它的代码如下所示:

```
tx = session.beginTransaction();
session.save(monkey);
tx.commit();
```

在 `test()` 方法中创建了一个 `Monkey` 对象, 还有一个 `homeAddress` 对象和 `comAddress` 对象, 建立了它们的关联关系, 然后调用 `saveMonkey()` 方法保存这个实例:

```
Monkey monkey=new Monkey();
Address homeAddress=new Address("province1","city1","street1",
    "100001",monkey);
Address comAddress=new Address("province2","city2","street2",
    "200002",monkey);
monkey.setName("Tom");
monkey.setHomeAddress(homeAddress);
monkey.setComAddress(comAddress);

saveMonkey(monkey);
```

`Session` 的 `save()` 方法向 `MONKEYS` 表插入一条记录, 同时还会向 `ADDRESSES` 表插入两条记录, 执行如下 `insert` 语句:

```
insert into ADDRESSES(ID,CITY,STREET,PROVINCE,ZIPCODE)
    values (1, 'city1', 'street1', 'province1', '100001 ');
insert into ADDRESSES(ID,CITY,STREET,PROVINCE,ZIPCODE)
    values (2, 'city2', 'street2', 'province2', '200002');
insert into MONKEYS (ID,NAME, HOME_ADRESS_ID,COM_ADDRESS_ID)
    values (1, 'Tom', 1,2);
```

(2) 运行 `loadMonkey()` 方法, 它的代码如下所示:

```
tx = session.beginTransaction(); //第 1 行
Monkey monkey=(Monkey)session.get(Monkey.class,id);//第 2 行
Hibernate.initialize(monkey.getHomeAddress());//第 3 行
Hibernate.initialize(monkey.getComAddress());//第 4 行
tx.commit(); //第 5 行
return monkey; //第 6 行
```

在默认情况下, 多对一关联采用延迟检索策略。因此程序第 2 行仅执行查询 `MONKEYS` 表的 `select` 语句:

```
select * from MONKEYS where ID=1;
```

在默认情况下，一对一关联采用迫切左外连接检索策略。因此，程序第 3 行执行以下 select 语句：

```
select a.ID, a.CITY, a.PROVINCE, a.STREET, a.ZIPCODE,
m.ID,m.NAME, m.HOME_ADDRESS_ID, m.COM_ADDRESS_ID
from ADDRESSES a left outer join MONKEYS m
on a.ID=m.HOME_ADDRESS_ID where a.ID=1;

select * from MONKEYS where HOME_ADDRESS_ID=1;
```

程序第 4 行执行以下 select 语句：

```
select a.ID, a.CITY, a.PROVINCE, a.STREET, a.ZIPCODE,
m.ID,m.NAME, m.HOME_ADDRESS_ID, m.COM_ADDRESS_ID
from ADDRESSES a left outer join MONKEYS m
on a.ID=m.HOME_ADDRESS_ID where a.ID=2
```

由于在 Address.hbm.xml 文件中指定与 Address 类的 monkey 属性关联的是 Monkey 类的 homeAddress 属性：

```
<one-to-one name="monkey"
class="mypack.Monkey"
property-ref="homeAddress"
/>
```

因此，程序第 4 行执行的 select 语句中的左外连接条件为：“a.ID=c.HOME\_ADDRESS\_ID”。以上 select 语句的查询结果为：

ID	CITY	PROVINCE	STREET	ZIPCODE	ID	NAME	HOME_ADDRESS_ID	COM_ADDRESS_ID
2	city2	province2	street2	200002	NULL	NULL	NULL	NULL

(3) 运行 printMonkey()方法，它的代码如下所示：

```
//从 Monkey 对象导航到 homeAddress 对象
Address homeAddress=monkey.getHomeAddress();
//从 Monkey 对象导航到 comAddress 对象
Address comAddress=monkey.getComAddress();
System.out.println("Home Address of "+monkey.getName()+" is: "
+homeAddress.getProvince()+" "
+homeAddress.getCity()+" "
+homeAddress.getStreet());

System.out.println("Company Address of "+monkey.getName()+" is: "
+comAddress.getProvince()+" "
+comAddress.getCity()+" "
+comAddress.getStreet());

//从 homeAddress 对象导航到 Monkey 对象
if(homeAddress.getMonkey()==null)
```

```

System.out.println("Can not naviagte from homeAddress to Monkey.");

//从 comAddress 对象导航到 Monkey 对象，导航失败
if(comAddress.getMonkey()==null)
    System.out.println("Can not naviagte from comAddress to Monkey.");

```

以上程序的输出结果为：

```

Home Address of Tom is: provincel city1 street1
Company Address of Tom is: province2 city2 street2
Can not naviagte from comAddress to Monkey.

```

由于在 Monkey.hbm.xml 和 Address.hbm.xml 文件中仅映射了从 homeAddress 对象到 Monkey 对象的双向关联，因此通过 homeAddress.getMonkey()方法能够从 homeAddress 对象导航到 Monkey 对象，而 comAddress.getMonkey()方法返回 null。

### 13.1.2 按照主键映射

如图 13-3 所示，在 Monkey 类中只有一个 address 属性，那么 Monkey 类与 Address 类之间只存在一个一对一关联关系，在这种情况下可以考虑按照主键映射方式。

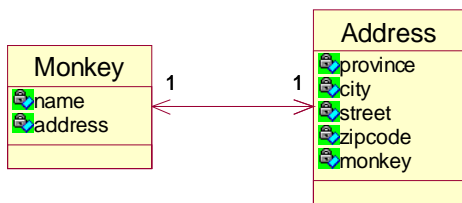


图 13-3 Monkey 类与 Address 类之间只存在一个一对一关联关系

在图 13-4 中，ADDRESSES 表的 ID 字段既是主键，同时作为外键参照 MONKEYS 表的主键，也就是说，ADDRESSES 表与 MONKEYS 表共享主键。

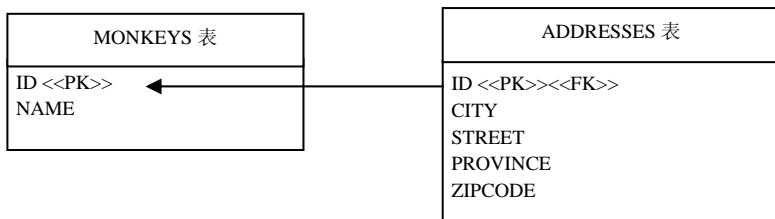


图 13-4 MONKEYS 表和 ADDRESSES 表的结构

在 Monkey.hbm.xml 文件中，用<one-to-one>元素来映射 Monkey 类的 address 属性：

```

<one-to-one name="address"
    class="mypack.Address"

```



```
        cascade="all"
    />
```

以上<one-to-one>元素的 cascade 属性为 all, 表明当保存、更新或删除 Monkey 对象时, 会级联保存、更新或删除 address 对象。

在 Address.hbm.xml 文件中, 用<one-to-one>元素来映射 Address 类的 monkey 属性:

```
<one-to-one name="monkey"
    class="mypack.Monkey"
    constrained="true"
/>
```

<one-to-one>元素的 constrained 属性为 true, 表明 ADDRESSES 表的 ID 主键同时作为外键参照 MONKEYS 表。在 Address.hbm.xml 文件中, 必须为 OID 使用 foreign 标识符生成策略:

```
<id name="id" type="long" column="ID">
    <generator class="foreign">
        <param name="property">monkey</param>
    </generator>
</id>
```

如果使用了 foreign 标识符生成策略, Hibernate 就会保证 Address 对象与关联的 Monkey 对象共享同一个 OID。

本节的范例程序位于配套光盘的 sourcecode\chapter13\13.1.2 目录下, 运行该程序前, 需要在 SAMPLEDB 数据库中手工创建 MONKEYS 和 ADDRESSES 表, 相关的 SQL 脚本文件为\13.1.2\schema\sampladb.sql。

在 DOS 命令行下进入 chapter13 根目录, 然后输入命令:

```
ant -file build1.2.xml run
```

就会运行 BusinessService 类。BusinessService 类的源程序的结构和本章 13.1.1 节的例程 13-1 很相似。BusinessService 的 main()方法调用 test()方法, test()方法依次调用以下方法。

- saveMonkey(): 保存一个 Monkey 对象。
- loadMonkey(): 加载一个 Monkey 对象。
- printMonkey(): 打印 Monkey 对象的信息, 包括它的 address 信息。

(1) 运行 saveMonkey (Monkey monkey)方法。在 test()方法中创建了一个 Monkey 对象, 还有一个 address 对象, 建立了它们的关联关系, 然后调用 saveMonkey()方法保存这个实例:

```
Monkey monkey=new Monkey();
```

```

Address address=new Address("provincel","city1","street1","100001",
    monkey);
monkey.setName("Tom");
monkey.setAddress(address);
saveMonkey(monkey);

```

Session 的 save()方法向 MONKEYS 表插入一条记录,同时还会向 ADDRESSES 表插入一条记录,执行如下 insert 语句:

```

insert into MONKEYS (ID,NAME)values (1, 'Tom');
insert into ADDRESSES(ID,CITY,STREET,PROVINCE,ZIPCODE)
    values (1, 'city1', 'street1', 'provincel', '100001 ');

```

(2) 运行 loadMonkey()方法:

```

tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,id);
tx.commit();
return monkey;

```

在默认情况下, Hibernate 对一对一关联采用迫切左外连接检索策略, Hibernate 执行以下 select 语句:

```

select m.ID,m.NAME, a.ID,a.CITY,a.STREET,a.PROVINCE,a.ZIPCODE
from MONKEYS m
left outer join ADDRESSES a on m.ID=a.ID
where m.ID=1;

```

(3) 运行 printMonkey()方法, 它的代码如下所示:

```

//从 Monkey 对象导航到 address 对象
Address address=monkey.getAddress();
System.out.println("Address of "+monkey.getName()+" is: "
    +address.getProvince()+" "
    +address.getCity()+" "
    +address.getStreet());

//从 address 对象导航到 Monkey 对象
if(address.getMonkey()==null)
    System.out.println("Can not naviagte from address to Monkey.");

```

以上程序的输出结果为:

```
Address of Tom is: provincel city1 street1
```

由于在 Monkey.hbm.xml 和 Address.hbm.xml 文件中映射了 Address 对象和 Monkey 对象的双向关联,因此通过 address.getMonkey()方法能够从 Address 对象导航到 Monkey 对象。

## 13.2 映射单向多对多关联

假定一个猴子可以拜多个武术师（假定武术师都是来自天届的各路神仙，非花果山的猴子）为师，一个武术师可以教多个猴子。因此 **Monkey** 类与 **Teacher** 类（代表武术师）之间为多对多关联关系。

假定仅建立了从 **Monkey** 类到 **Teacher** 类的单向多对多关联。在 **Monkey** 类中需要定义集合类型的 **teachers** 属性，而在 **Teacher** 类中不需要定义集合类型的 **monkeys** 属性。图 13-5 显示了 **Monkey** 类和 **Teacher** 类的关联关系。

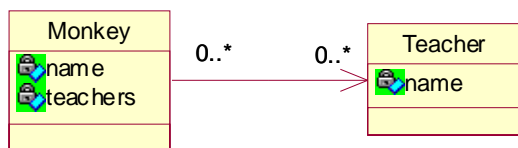


图 13-5 **Monkey** 类与 **Teacher** 类的单向一对多关联关系

在 **Monkey** 类中定义 **teachers** 属性的代码如下所示：

```

private Set teachers=new HashSet();
public Set getTeachers() {
    return this.teachers;
}
public void setTeachers(Set teachers) {
    this.teachers = teachers;
}
  
```

在关系数据模型中，无法直接表达 **MONKEYS** 表和 **TEACHERS** 表之间的多对多关系，需要创建一个连接表 **LEARNING**，它同时参照 **MONKEYS** 表和 **TEACHERS** 表。图 13-6 显示了这 3 张表的结构。

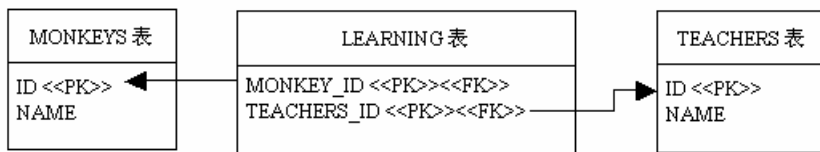


图 13-6 **MONKEYS** 表、**TEACHERS** 表及连接表的结构

**LEARNING** 表以 **MONKEY\_ID** 和 **TEACHER\_ID** 作为联合主键，此外，**MONKEY\_ID** 字段作为外键参照 **MONKEYS** 表，而 **TEACHER\_ID** 字段作为外键参照 **TEACHERS** 表。以下是 **LEARNING** 表的 DDL 定义：

```

create table LEARNING(
    MONKEY_ID bigint not null,
    TEACHER_ID bigint not null,
  
```

```

        primary key(MONKEY_ID,TEACHER_ID)
    );

    alter table LEARNING add index IDX_MONKEY(MONKEY_ID),
    add constraint FK_MONKEY foreign key (MONKEY_ID) references
        MONKEYS(ID);

    alter table LEARNING add index IDX_TEACHER(TEACHER_ID),
    add constraint FK_TEACHER foreign key (TEACHER_ID) references
        TEACHERS(ID);

```

在 Monkey.hbm.xml 文件中, 映射 Monkey 类的 teachers 属性的代码如下所示:

```

<set name="teachers" table="LEARNING"
    lazy="true"
    cascade="save-update">
    <key column="MONKEY_ID" />
    <many-to-many class="mypack.Teacher" column="TEACHER_ID" />
</set>

```

<set>元素的 cascade 属性为“save-update”, 表明保存或更新 Monkey 对象时, 会级联保存或更新与它关联的 Teacher 对象。<set>元素的<key>子元素指定 LEARNING 表中参照 MONKEYS 表的外键为 MONKEY\_ID, <many-to-many>子元素的 class 属性指定 teachers 集合中存放的是 Teacher 对象, column 属性指定 LEARNING 表中参照 TEACHERS 表的外键为 TEACHER\_ID。

#### Tips

对于多对多关联, cascade 属性设为“save-update”是合理的, 但是不允许把 cascade 属性设为“all”、“delete”或“all-delete-orphans”。假如删除一个 Monkey 对象时, 还级联删除与它关联的所有 Teacher 对象, 由于这些 Teacher 对象有可能还与其他 Monkey 对象关联, 因此当 Hibernate 执行级联删除时, 会违反数据库的外键参照完整性。

本节的范例程序位于配套光盘的 sourcecode\chapter13\13.2 目录下, 运行该程序前, 需要在 SAMPLEDB 数据库中手工创建 MONKEYS、TEACHERS 和 LEARNING 表, 相关的 SQL 脚本文件为\13.2\schema\sampldb.sql。

在 DOS 命令行下进入 chapter13 根目录, 然后输入命令:

```
ant -file build2.xml run
```

就会运行 BusinessService 类。BusinessService 类的源程序的结构和本章 13.1.1 节的例程 13-1 很相似。BusinessService 的 main()方法调用 test()方法, test()方法依次调用以下方法。

- saveMonkey(): 保存一个 Monkey 对象。
- loadMonkey(): 加载一个 Monkey 对象。

- `printMonkey ()`: 打印 `Monkey` 对象的信息, 包括它的 `teachers` 集合中的所有 `Teacher` 对象的信息。

(1) 运行 `saveMonkey (Monkey monkey)` 方法。在 `test()` 方法中创建了两个 `Monkey` 对象和两个 `Teacher` 对象, 建立了它们的关联关系, 然后调用 `saveMonkey()` 方法保存 `Monkey` 对象:

```
Teacher teacher1=new Teacher("二郎神");
Teacher teacher2=new Teacher("红孩儿");

Monkey monkey1=new Monkey();
monkey1.setName("智多星");
monkey1.getTeachers().add(teacher1);
monkey1.getTeachers().add(teacher2);

Monkey monkey2=new Monkey();
monkey2.setName("老顽童");
monkey2.getTeachers().add(teacher1);

saveMonkey(monkey1);
saveMonkey(monkey2);
```

图 13-7 显示了以上程序建立的 `Monkey` 对象与 `Teacher` 对象的关联关系。

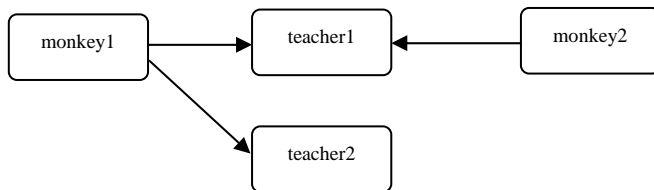


图 13-7 `Monkey` 对象与 `Teacher` 对象的关联关系

当 `Session` 的 `save()` 方法保存 `monkey1` 对象时, 向 `MONKEYS` 表插入一条记录, 同时还会分别向 `TEACHERS` 和 `LEARNING` 表插入两条记录, 执行如下 `insert` 语句:

```
insert into MONKEYS (ID,NAME) values (1, '智多星');
insert into TEACHERS(ID,NAME) values (1,'二郎神');
insert into TEACHERS(ID,NAME) values (2,'红孩儿');
insert into LEARNING(MONKEY_ID,TEACHER_ID) values(1,1);
insert into LEARNING(MONKEY_ID,TEACHER_ID) values(1,2);
```

当 `Session` 的 `save()` 方法保存 `monkey2` 对象时, 向 `MONKEYS` 表插入一条记录, 同时向 `LEARNING` 表插入一条记录, 由于 `<set>` 元素的 `cascade` 属性为 `save-update`, 并且与 `monkey2` 对象关联的 `teacher1` 对象已经被保存到数据库中, 因此会更新数据库中的 `teacher1` 对象。Hibernate 执行如下 SQL 语句:

```
insert into MONKEYS (ID,NAME) values (2, '老顽童');
```

```
insert into LEARNING(MONKEY_ID,TEACHER_ID) values(2,1);
update TEACHERS set NAME='二郎神' where ID=1;
```

### Tips

teacher1 对象的属性没有任何变化,为什么 Hibernate 还会更新这个 teacher1 对象呢?这是因为当 Session 保存 monkey2 对象时,teacher1 对象已经变成了游离对象,在当前 Session 对象的缓存中没有原来 teacher1 对象的快照,Hibernate 无法知道当前 teacher1 对象是否和数据库中的数据保持一致,因此会执行以上 update 语句。

(2) 运行 loadMonkey()方法,它的代码如下所示:

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,id);
Hibernate.initialize(monkey.getTeachers());
tx.commit();
return monkey;
```

由于在 Monkey.hbm.xml 文件中对 teachers 集合使用了延迟检索策略,因此必须通过 Hibernate 类的 initialize()方法显示初始化 teachers 集合,这样才能保证当 Monkey 对象成为游离对象后,BusinessService 类的 test()方法能够正常访问 teachers 集合中的元素。Hibernate 类的 initialize()方法执行以下 select 语句:

```
select le.MONKEY_ID, le.TEACHER_ID, t.ID,t.NAME
from LEARNING le left outer join TEACHERS t
on le.TEACHER_ID=t.ID where le.MONKEY_ID=1;
```

(3) 运行 printMonkey()方法,它的代码如下所示:

```
Set teachers=monkey.getTeachers();
Iterator it=teachers.iterator();
while(it.hasNext()){
    Teacher teacher=(Teacher)it.next();
    System.out.println(monkey.getName()+" "+teacher.getName());
}
```

以上程序的输出结果为:

```
智多星 红孩儿
智多星 二郎神
```

## 13.3 映射双向多对多关联关系

对于 Monkey 类与 Teacher 类的双向多对多关联,下面介绍 3 种映射方式:

- 关联的两端都使用<set>元素,把其中一端的 inverse 属性设为 true,参见 13.3.1 节。
- 当关联本身包含属性,可以定义专门的组件类 Learning 来描述关联。在

Monkey.hbm.xml 和 Teacher.hbm.xml 文件的 <set> 元素中，用 <composite-element> 子元素来映射 Learning 组件类参见 13.3.2 节。

- 把多对多关联关系分解为 Monkey 与 Learning 类，以及 Teacher 与 Learning 类的一对多关联关系。参见 13.3.3 节。

### 13.3.1 关联两端使用<set>元素

假定建立了从 Monkey 类到 Teacher 类的双向多对多关联。在 Monkey 类中需要定义集合类型的 teachers 属性，并且在 Teacher 类也需要定义集合类型的 monkeys 属性。图 13-8 显示了 Monkey 类和 Teacher 类的关联关系。

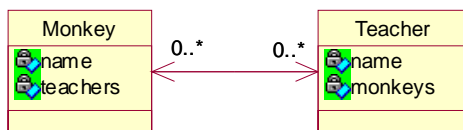


图 13-8 Monkey 与 Teacher 的双向多对多关联关系

MONKEYS 表、TEACHERS 表和 LEARNING 表的结构和本章 13.2 节的图 13-6 一样。在 Monkey.hbm.xml 文件中，映射 Monkey 类的 teachers 属性的代码如下所示：

```

<set name="teachers" table="LEARNING"
    lazy="true"
    cascade="save-update">
    <key column="MONKEY_ID" />
    <many-to-many class="mypack.Teacher" column="TEACHER_ID" />
</set>

```

在 Teacher.hbm.xml 文件中，映射 Teacher 类的 monkeys 属性的代码如下所示：

```

<set name="monkeys" table="LEARNING"
    lazy="true"
    inverse="true"
    cascade="save-update">
    <key column="TEACHER_ID" />
    <many-to-many class="mypack.Monkey" column="MONKEY_ID" />
</set>

```

对于双向多对多关联的两端，必须把其中一端的<set>元素的 inverse 属性设为“true”。在 BusinessService 类中，必须同时建立从 Monkey 到 Teacher，以及从 Teacher 到 Monkey 的关联关系：

```

Teacher teacher1=new Teacher("二郎神");
Teacher teacher2=new Teacher("红孩儿");

Monkey monkey1=new Monkey();
monkey1.setName("智多星");

```

```

monkey1.getTeachers().add(teacher1);
monkey1.getTeachers().add(teacher2);
teacher1.getMonkeys().add(monkey1);
teacher2.getMonkeys().add(monkey1);

Monkey monkey2=new Monkey();
monkey2.setName("老顽童");
monkey2.getTeachers().add(teacher1);
teacher1.getMonkeys().add(monkey2);

```

图 13-9 显示了以上程序建立的 Monkey 对象与 Teacher 对象的关联关系。

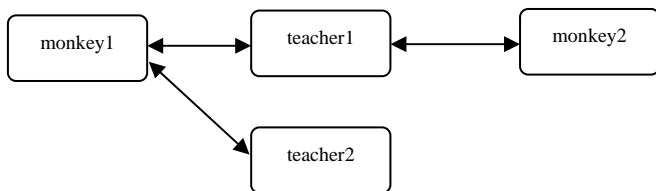


图 13-9 Monkey 对象与 Teacher 对象的关联关系

本节的范例程序位于配套光盘的 sourcecode\chapter13\13.3.1 目录下，运行该程序前，需要在 SAMPLEDB 数据库中手工创建 MONKEYS、TEACHERS 和 LEARNING 表，相关的 SQL 脚本文件为\13.3.1\schema\sampladb.sql。

在 DOS 命令行下进入 chapter13 根目录，然后输入命令：

```
ant -file build3.1.xml run
```

就会运行 BusinessService 类，它的运行结果和本章 13.2 节的 BusinessService 类的运行结果相似，因此不再做详细介绍。

### 13.3.2 使用组件类集合

每个猴子都会向武术师学习一些功夫，如向二郎神学习七十三变的功夫，向红孩儿学习三昧真火的功夫。这意味着对于 Teacher 与 Monkey 之间的关联关系，关联本身也具有一个代表功夫的 gongfu 属性。例如对于猴子智多星与二郎神之间的关联关系，关联本身的 gongfu 属性的值为“七十三变”。可以通过专门的组件类 Learning 来描述 Monkey 与 Teacher 类的关联信息。图 13-10 为 Monkey、Teacher 和 Learning 类的类框图。

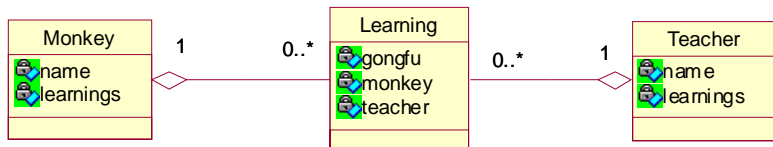


图 13-10 Monkey、Teacher 和 Learning 类的类框图



例程 13-2 是 Learning 类的源程序，Learning 类作为组件类没有 OID。

例程 13-2 Learning.java

```
public class Learning {
    private Teacher teacher;
    private Monkey monkey;
    private String gongfu;

    public Learning(Teacher teacher,Monkey monkey,String gongfu) {
        this.teacher= teacher;
        this.monkey = monkey;
        this.gongfu=gongfu;
    }

    public Learning() {}

    //省略显示 teacher、monkey 和 gongfu 属性的 getXXX() 和 setXXX() 方法
    .....
}
```

在 Monkey 类和 Teacher 类中都定义了 Set 类型的 learnings 属性：

```
private Set learnings=new HashSet();
public Set getLearnings() {
    return this.learnings;
}
public void setLearnings(Set learnings) {
    this.learnings = learnings;
}
```

在关系数据模型中，用 LEARNINGS 表作为连接表，图 13-11 显示 MONKEYS、TEACHERS 及 LEARNINGS 表的结构。

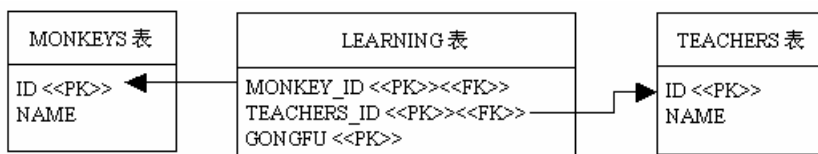


图 13-11 MONKEYS 表、TEACHERS 表及连接表的结构

LEARNINGS 表以所有的字段作为联合主键，此外，它的 MONKEY\_ID 字段作为外键参照 MONKEYS 表，而 TEACHER\_ID 字段作为外键参照 TEACHERS 表。以下是 LEARNINGS 表的 DDL 定义：

```
create table LEARNING(
    MONKEY_ID bigint not null,
    TEACHER_ID bigint not null,
    GONGFU varchar(15),
```

```

        primary key(MONKEY_ID,TEACHER_ID,GONGFU)
    );

    alter table LEARNINGS add index IDX_MONKEY(MONKEY_ID),
    add constraint FK_MONKEY foreign key (MONKEY_ID) references
        MONKEYS(ID);

    alter table LEARNINGS add index IDX_TEACHER(TEACHER_ID),
    add constraint FK_TEACHER foreign key (TEACHER_ID) references
        TEACHERS(ID);

```

例程 13-3 是 Monkey.hbm.xml 文件的源代码。

例程 13-3 Monkey.hbm.xml

```

<hibernate-mapping >

    <class name="mypack.Monkey" table="MONKEYS" >
        <id name="id" type="long" column="ID">
            <generator class="increment"/>
        </id>

        <property name="name" column="NAME" type="string" />

        <set name="learnings" lazy="true" table="LEARNING" >
            <key column="MONKEY_ID" />

            <composite-element class="mypack.Learning" >
                <parent name="monkey" />
                <many-to-one name="teacher" class="mypack.Teacher"
                    column="TEACHER_ID" not-null="true"/>

                <property name="gongfu" column="GONGFU"
                    type="string" not-null="true" />

            </composite-element>
        </set>

    </class>

</hibernate-mapping>

```

Monkey 类的 learnings 属性用 <set> 元素映射。该 <set> 元素中的 <composite-element> 元素用于映射 Learning 组件类, 它的所有属性都不允许为 null。在 Teacher.hbm.xml 文件中按同样方式映射 Teacher 类的 learnings 集合:

```

<set name="learnings" lazy="true" inverse="true" table="LEARNING" >

```

```

<key column="TEACHER_ID" />
<composite-element class="mypack.Learning" >
  <parent name="teacher" />
  <many-to-one name="monkey" class="mypack.Monkey"
    column="MONKEY_ID" not-null="true"/>
  <property name="gongfu" column="GONGFU"
    type="string" not-null="true" />
</composite-element>
</set>

```

本节的范例程序位于配套光盘的 sourcecode\chapter13\13.3.2 目录下,运行该程序前,需要在 SAMPLEDB 数据库中手工创建 MONKEYS、TEACHERS 和 LEARNINGS 表,相关的 SQL 脚本文件为 13.3.2\schema\sampledb.sql。

在 DOS 命令行下进入 chapter13 根目录,然后输入命令:

```
ant -file build3.2.xml run
```

就会运行 BusinessService 类。BusinessService 类的源程序的结构和本章 13.1.1 节的例程 13-1 很相似。BusinessService 的 main()方法调用 test()方法, test()方法依次调用以下方法:

- saveTeacher(): 保存一个 Teacher 对象。
- saveMonkey(): 保存一个 Monkey 对象。
- loadMonkey(): 加载一个 Monkey 对象。
- printMonkey(): 打印 Monkey 对象的信息,包括它的 learnings 集合中的所有 Learning 对象的信息。

(1) 运行 saveTeacher (Teacher teacher)方法。在 test()方法中创建了两个 Teacher,然后调用 saveTeacher()方法保存 Teacher 对象:

```

Teacher teacher1=new Teacher("二郎神",null);
Teacher teacher2=new Teacher("红孩儿",null);
saveTeacher(teacher1);
saveTeacher(teacher2);

```

(2) 运行 saveMonkey (Monkey monkey)方法。在 test()方法中创建了一个 Monkey 对象和两个 Learning 对象,建立了它们的关联关系,然后调用 saveMonkey()方法保存 Monkey 对象:

```

Monkey monkey=new Monkey();
monkey.setName("智多星");
Learning learning1=new Learning(teacher1,monkey,"七十三变");
Learning learning2=new Learning(teacher2,monkey,"三昧真火");

monkey.getLearnings().add(learning1);
monkey.getLearnings().add(learning2);

```

```
saveMonkey(monkey);
```

当 Session 的 save() 方法保存 Monkey 对象时, 向 MONKEYS 表插入一条记录, 同时向 LEARNINGS 表插入两条记录。LEARNINGS 表中包含以下数据:

MONKEY_ID	TEACHER_ID	GONGFU
1	1	七十二变
1	2	三昧真火

(3) 运行 loadMonkey() 方法, 它的代码如下所示:

```
tx = session.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,id);

Set learnings=monkey.getLearnings();
Iterator it=learnings.iterator(); //初始化 Learnings 集合
while(it.hasNext()){
    Learning learning=(Learning)it.next();
    //初始化与 Learning 关联的 Teacher 对象
    Hibernate.initialize(learning.getTeacher());
}
tx.commit();

return monkey;
```

Session 加载 Monkey 对象时, 执行以下 select 语句:

```
select ID, NAME from MONKEYS where ID=1;
```

由于在 Monkey.hbm.xml 文件中对 learnings 集合使用了延迟检索策略, 因此必须初始化 learnings 集合, 以及初始化与该集合中的 Learning 对象关联的 Teacher 对象, 这样才能保证当 Monkey 对象成为游离对象后, BusinessService 类的 test() 方法能够正常访问 learnings 集合中的元素。

(4) 运行 printMonkey() 方法, 它的代码如下所示:

```
System.out.println("名字:"+monkey.getName());

Set learnings=monkey.getLearnings();
Iterator it=learnings.iterator();
while(it.hasNext()){
    Learning learning=(Learning)it.next();
    System.out.println("-----");
    System.out.println("老师:"+learning.getTeacher().getName());
    System.out.println("功夫:"+learning.getGongfu());
}
```

以上程序的输出结果为：

```

名字:智多星
-----
老师:二郎神
功夫:七十三变
-----
老师:红孩儿
功夫:三昧真火
    
```

### 13.3.3 把多对多关联分解为两个一对多关联

对于 Monkey 与 Teacher 的多对多关联，也可以把它分解为两个一对多关联，如图 13-12 所示，Learning 为独立的实体类，有单独的 OID，Monkey 类与 Learning 类，以及 Teacher 类与 Learning 类都是一对多双向关联关系。

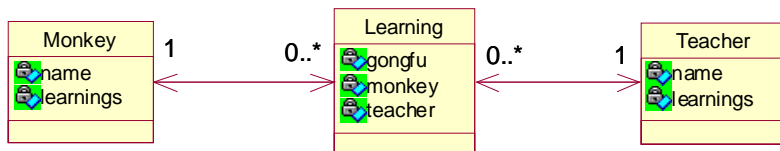


图 13-12 Monkey、Learning 与 Teacher 类的类框图

在关系数据模型中，用 LEARNINGS 表作为连接表，LEARNINGS 表有单独的 ID 代理主键。图 13-13 是 MONKEYS、TEACHERS 及连接表的结构。

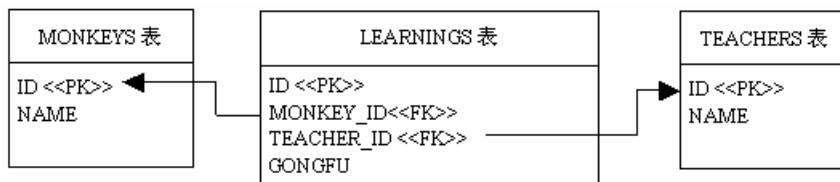


图 13-13 MONKEYS 表、TEACHERS 表及连接表的结构

以下是 LEARNINGS 表的 DDL 定义：

```

create table LEARNING(
    ID bigint not null,
    MONKEY_ID bigint not null,
    TEACHER_ID bigint not null,
    GONGFU varchar(15),
    primary key(ID)
);

alter table LEARNINGS add index IDX_MONKEY(MONKEY_ID),
add constraint FK_MONKEY foreign key (MONKEY_ID) references
    MONKEYS(ID);
    
```

```
alter table LEARNINGS add index IDX_TEACHER(TEACHER_ID),
add constraint FK_TEACHER foreign key (TEACHER_ID) references
TEACHERS(ID);
```

在 Monkey.hbm.xml 文件中, 映射 Monkey 类的 learnings 属性的代码如下所示:

```
<set name="learnings" lazy="true" inverse="true" cascade=
    "save-update">
    <key column="MONKEY_ID" />
    <one-to-many class="mypack.Learning" />
</set>
```

在 Teacher.hbm.xml 文件中按同样方式映射 Teacher 类的 learnings 集合:

```
<set name="learnings" lazy="true" inverse="true" cascade=
    "save-update">
    <key column="TEACHER_ID" />
    <one-to-many class="mypack.Learning" />
</set>
```

由于 Learning 类是实体类, 因此也必须为它创建 Learning.hbm.xml 文件, 例程 13-4 是它的源代码, 其中两个 <many-to-one> 元素分别用来映射 Learning 类的 monkey 属性和 teacher 属性。

例程 13-4 Learning.hbm.xml

```
<hibernate-mapping >

    <class name="mypack.Learning" table="LEARNING" >
        <id name="id" type="long" column="ID">
            <generator class="increment"/>
        </id>

        <property name="gongfu" column="GONGFU" type="string" />

        <many-to-one name="monkey" column="MONKEY_ID"
            class="mypack.Monkey" not-null="true" />

        <many-to-one name="teacher" column="TEACHER_ID"
            class="mypack.Teacher" not-null="true" />

    </class>
</hibernate-mapping>
```

本节的范例程序位于配套光盘的 sourcecode\chapter13\13.3.3 目录下, 运行该程序前, 需要在 SAMPLEDB 数据库中手工创建 MONKEYS、TEACHERS 和 LEARNINGS 表, 相关的 SQL 脚本文件为 13.3.3\schema\sampledb.sql。

在 DOS 命令行下进入 chapter13 根目录，然后输入命令：

```
ant -file build3.3.xml run
```

就会运行 **BusinessService** 类，它的运行结果和本章 13.3.2 节的 **BusinessService** 类的运行结果相似，因此不再做详细介绍。

## 13.4 小结

映射一对一关联有两种方式，如果 **Monkey** 与 **Address** 类之间有两个一对一关联，可以使用按外键映射方式，但这种映射方式只能映射 **Monkey** 与 **homeAddress** 对象的双向一对一关联，而不能同时映射 **Monkey** 与 **comAddress** 对象的双向一对一关联，从 **Monkey** 到 **comAddress** 对象为单向关联。如果 **Monkey** 与 **Address** 类之间只有一个一对一关联，应该优先考虑使用按主键映射方式。

对于双向多对多关联，必须把其中一端的 **inverse** 属性设为 **true**，关联的两端都可以使用 **<set>** 元素。

事实上，所有的多对多关联都可以分解为两个一对多关联，按照这种方式映射多对多关联，会使对象模型和关系数据模型具有更好的可扩展性。

现在，大家已经跟随悟空掌握了把对象模型中的各种关系映射到数据库的方法。这些关系包括：

- 一对多关联关系：参见第 5 章
- 一对一和多对多关联关系：参见本章
- 组成关系：参见第 8 章
- 继承关系：参见第 10 章

在映射每种关系时，读者需要从 4 个方面去把握：

- (1) 如何在持久化类的代码中体现出类与类之间的特定的关系
- (2) 如何把数据库表与持久化类对应。
- (3) 如何创建对象-关系映射文件。
- (4) 如何通过 **Hibernate API** 来操纵具有特定关系的多个对象。

## 第 14 章 声明数据库事务

花果山的智多星到银行办理转账业务，把 100 元钱转到小不点的账号上。银行按照智多星的要求，先从智多星账号上扣除 100 元。接下来在打算给小不点的账号上增加 100 元时，发现小不点的账号处于冻结状态，因此无法给小不点的账号增钱。显然，转账业务无法正常执行了。可是，智多星的账号上已经少了 100 元，怎么办呢？此时，就必须依靠数据库系统的事务管理机制，请求数据库撤销本次转账业务中涉及的所有数据库操作，使智多星的账号上的余额恢复到执行转账业务之前的状态。

转账业务中涉及的所有数据库操作可以看做是一个数据库事务。数据库事务是指由一个或者多个 SQL 语句组成的工作单元，这个工作单元中的 SQL 语句相互依赖，如果有一个 SQL 语句执行失败，就必须撤销整个工作单元。本章介绍了数据库事务的概念，并且介绍了在应用程序中声明事务边界的方法。

本章重点介绍通过 **Hibernate API** 来声明事务边界的方法。为了便于读者理解声明事务边界的原理，本章还简要介绍通过 **MySQL.exe** 客户程序，以及在程序中通过 **JDBC API** 来声明事务边界的过程。

### 14.1 数据库事务的概念

在现实生活中，事务是指一组相互依赖的操作行为，如银行交易、股票交易或网上购物。事务的成功取决于这些相互依赖的操作行为是否都能执行成功，只要有一个操作行为失败，就意味着整个事务失败。例如，智多星到银行办理转账事务，把 100 元钱转到小不点的账号上，这个事务包含以下操作行为：

- (1) 从智多星的账户上减去 100 元。
- (2) 往小不点的账户上增加 100 元。

显然，以上两个操作必须作为一个不可分割的工作单元。假如仅第一步操作执行成功，使得智多星的账户上扣除了 100 元，但是第二步操作执行失败，小不点的账户上没有增加 100 元，那么整个事务失败。

数据库事务是对现实生活中事务的模拟，它由一组在业务逻辑上相互依赖的 SQL 语句组成。假定 **ACCOUNTS** 表用于存放账户信息，它的数据如表 14-1 所示。



表 14-1 ACCOUNTS 表中的数据

ID	NAME	BALANCE
1	智多星	1000
2	小不点	1000

以上银行转账事务对应于以下 SQL 语句：

```
update ACCOUNTS set BALANCE=900 where ID=1;
update ACCOUNTS set BALANCE=1100 where ID=2;
```

这两条 SQL 语句只要有一条执行失败，ACCOUNTS 表中的数据就必须退回到最初的状态。如果两条 SQL 语句都执行成功，表示整个事务成功。图 14-1 显示了 ACCOUNTS 表中数据在转账事务中的状态转换图。

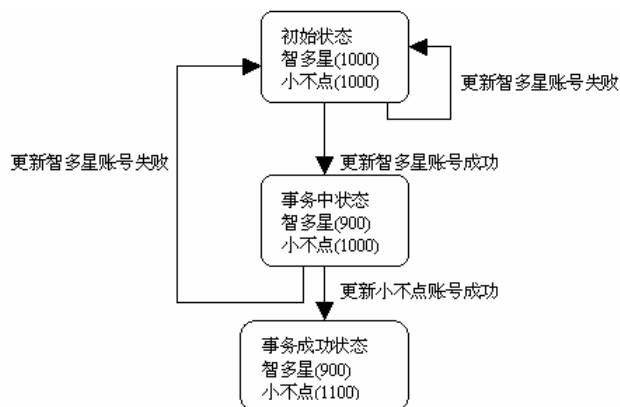


图 14-1 ACCOUNTS 表中数据在转账事务中的状态转换图

数据库事务必须具备 ACID 特征，ACID 是 Atomic（原子性）、Consistency（一致性）、Isolation（隔离性）和 Durability（持久性）的英文缩写。下面解释这几个特性的含义。

- 原子性：指整个数据库事务是不可分割的工作单元。只有事务中所有的操作执行成功，才算整个事务成功；事务中任何一个 SQL 语句执行失败，那么已经执行成功的 SQL 语句也必须撤销，数据库状态应该退回到执行事务前的状态。
- 一致性：指数据库事务不能破坏关系数据的完整性及业务逻辑上的一致性。例如对于银行转账事务，不管事务成功还是失败，应该保证事务结束后 ACCOUNTS 表中智多星和小不点的存款总额为 2000 元。
- 隔离性：指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。
- 持久性：指的是只要事务成功结束，它对数据库所做的更新就必须永久保存下来。即使发生系统崩溃，重新启动数据库系统后，数据库还能恢复到

事务成功结束时的状态。

事务的 **ACID** 特性是由关系数据库管理系统（**RDBMS**，在本书中也简称为数据库系统）来实现的。数据库管理系统采用日志来保证事务的原子性、一致性和持久性。日志记录了事务对数据库所做的更新，如果某个事务在执行过程中发生错误，就可以根据日志，撤销事务对数据库已做的更新，使数据库退回到执行事务前的初始状态。

数据库管理系统采用锁机制来实现事务的隔离性。当多个事务同时更新数据库中相同的数据时，只允许持有锁的事务能更新该数据，其他事务必须等待，直到前一个事务释放了锁，其他事务才有机会更新该数据。

## 14.2 声明事务边界的方式

数据库系统的客户程序只要向数据库系统声明了一个事务，数据库系统就会自动保证事务的 **ACID** 特性。声明事务包含以下内容：

- 事务的开始边界（**BEGIN**）。
- 事务的正常结束边界（**COMMIT**）：提交事务，永久保存被事务更新后的数据库状态。
- 事务的异常结束边界（**ROLLBACK**）：撤销事务，使数据库退回到执行事务前的初始状态。

图 14-2 显示了事务的生命周期。当一个事务开始后，要么以提交事务结束，要么以撤销事务结束。

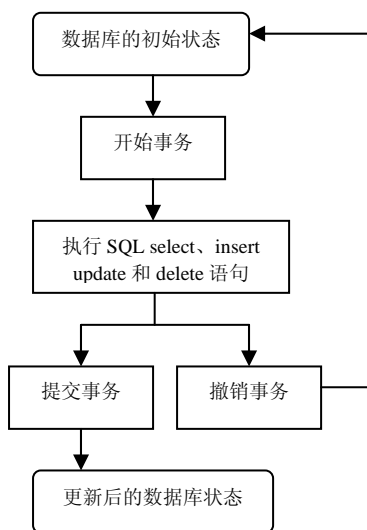


图 14-2 数据库事务的生命周期

数据库系统支持以下两种事务模式：

- 自动提交模式：每个 SQL 语句都是一个独立的事务，当数据库系统执行完一个 SQL 语句后，会自动提交事务。
- 手工提交模式：必须由数据库的客户程序显式指定事务开始边界和结束边界。

在 MySQL 中，数据库表分为 3 种类型：INNODB、BDB 和 MyISAM 类型。其中 InnoDB 和 BDB 类型的表支持数据库事务，而 MyISAM 类型的表不支持事务。在 MySQL 中用 create table 语句新建的表默认为 MyISAM 类型。如果希望创建 INNODB 类型的表，可以采用以下形式的 DDL 语句：

```
create table ACCOUNTS (
    ID bigint not null,
    NAME varchar(15),
    BALANCE decimal(10,2),
    primary key (ID)
) type=INNODB;
```

对于已存在的表，可以采用以下形式的 DDL 语句修改它的表类型：

```
alter table ACCOUNTS type=INNODB;
```

图 14-3 以 MySQL 为例，列出了它的两个客户程序：mysql.exe 程序和 Java 应用程序。

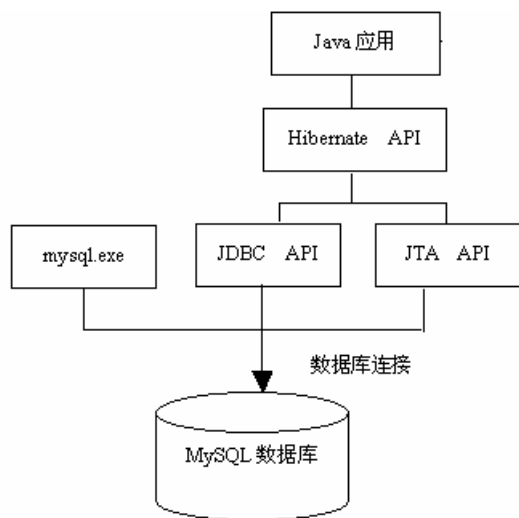


图 14-3 MySQL 数据库系统的客户程序

在图 14-3 中，mysql.exe 是 MySQL 软件自带的 DOS 命令行客户程序。对于 Java 应用，声明事务有以下方式：

- 方式一：直接通过 JDBC API 来声明事务。

- 方式二：直接通过 JTA API 来声明事务。JTA (Java Transaction API) 是 SUN 公司制定的标准事务 API，JTA 支持分布式的事务。由于本书的重点是介绍 Hibernate，因此没有详细介绍这种声明事务的方式。
- 方式三：直接通过 Hibernate API 来声明事务。Hibernate 封装了 JDBC API 和 JTA API，提供了统一的事务声明接口。应用程序通过 Hibernate API，而不是直接通过 JDBC API 或 JTA API 来声明事务，这有利于跨平台开发。

### 14.3 在mysql.exe程序中声明事务

每启动一个 mysql.exe 程序，就会得到一个单独的数据库连接。每个数据库连接都有一个全局变量 @@autocommit，表示当前的事务模式，它有两个可选值：

- 0：表示手工提交模式。
- 1：默认值，表示自动提交模式。

如果要查看当前的事务模式，可使用如下 SQL 命令：

```
mysql> select @@autocommit
```

如果要把当前的事务模式改为手工提交模式，可使用如下 SQL 命令：

```
mysql> set autocommit=0;
```

#### 1. 在自动提交模式下运行事务

在自动提交模式下，每个 SQL 语句都是一个独立的事务。如果在一个 mysql.exe 程序中执行 SQL 语句：

```
mysql> insert into ACCOUNTS values(1, '智多星', 1000);
```

MySQL 会自动提交这个事务，这意味着向 ACCOUNTS 表中新插入的记录会永久保存在数据库中。此时在另一个 mysql.exe 程序中执行 SQL 语句：

```
mysql> select * from ACCOUNTS;
```

这条 select 语句会查询到 ID 为 1 的 ACCOUNTS 记录。这表明在第一个 mysql.exe 程序中插入的 ACCOUNTS 记录被永久保存，这体现了事务的 ACID 特性中的永久性。

#### 2. 在手工提交模式下运行事务

在手工提交模式下，必须显式指定事务开始边界和结束边界。

- 事务的开始边界：begin
- 提交事务：commit
- 撤销事务：rollback

下面举例说明如何在手工提交模式下声明事务，步骤如下。

(1) 打开两个 DOS 控制台，分别转到 MySQL 根目录的 bin 子目录下，分别运行 mysql.exe 程序，在两个程序中都执行以下命令，以便设定手工提交事务模式：

```
mysql>set autocommit=0;
```

(2) 在两个程序中都执行以下命令，转到 SAMPLEDB 数据库：

```
mysql>use sampled;
```

(3) 在第一个 mysql.exe 程序中执行 SQL 语句：

```
mysql>begin;
mysql>insert into ACCOUNTS values(2, '小不点',1000);
```

(4) 在第二个 mysql.exe 程序中执行 SQL 语句：

```
mysql>begin;
mysql>select * from ACCOUNTS;
mysql>commit;
```

如图 14-4 所示，以上 select 语句的查询结果中并不包含 ID 为 2 的 ACCOUNTS 记录，这是因为第一个 mysql.exe 程序还没有提交事务。

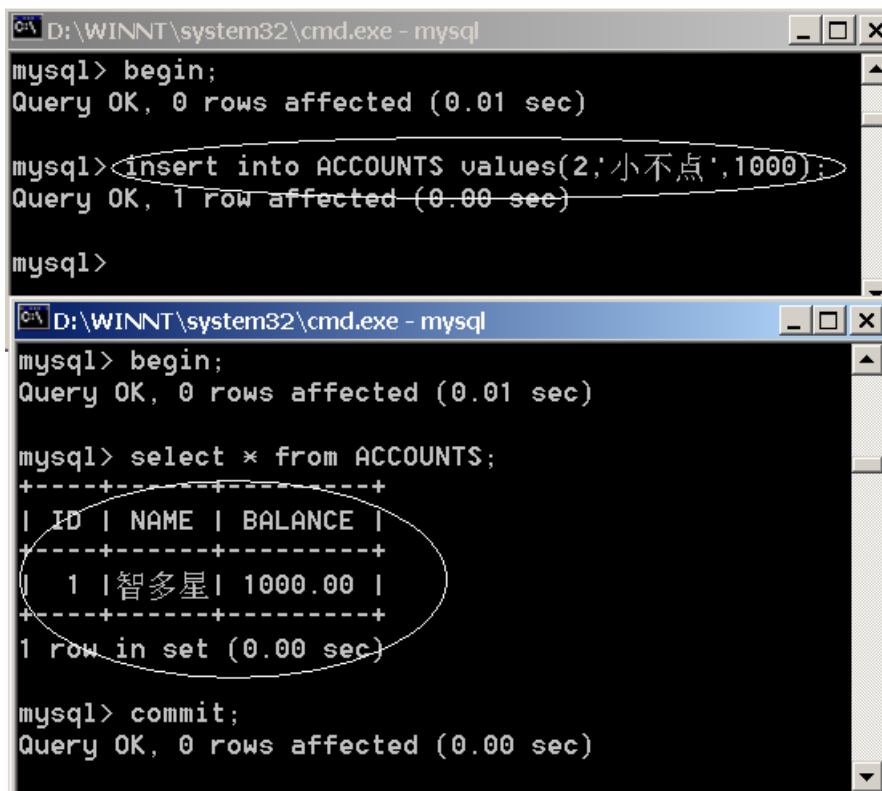


图 14-4 两个客户程序各自执行事务

(5) 在第一个 `mysql.exe` 程序中执行以下 SQL 语句，从而提交事务：

```
mysql>commit;
```

(6) 在第二个 `mysql.exe` 程序中执行 SQL 语句：

```
mysql>begin;
mysql>select * from ACCOUNTS;
mysql>commit;
```

此时，`select` 语句的查询结果中会包含 ID 为 2 的 `ACCOUNTS` 记录，这是因为第一个 `mysql.exe` 程序已经提交事务。

(7) 在第一个 `mysql.exe` 程序中执行 SQL 语句：

```
mysql>begin;
mysql>delete from ACCOUNTS;
mysql>commit;
mysql>begin;
mysql>insert into ACCOUNTS values(1, '智多星',1000);
mysql>insert into ACCOUNTS values(2, '小不点',1000);
mysql>rollback;
mysql>begin;
mysql>select * from ACCOUNTS;
mysql>commit;
```

以上 SQL 语句共包含 3 个事务：第一个事务删除 `ACCOUNTS` 表中所有的记录，然后提交该事务；第二个事务以撤销结束，因此它向 `ACCOUNTS` 表插入的两条记录不会被永久保存到数据库中；第三个事务的 `select` 语句查询结果为空。

## 14.4 Java应用通过JDBC API声明事务

在 JDBC API 中，`java.sql.Connection` 类代表一个数据库连接。以下程序用于创建一个 `Connection` 类的实例：

```
//加载 MySQL 驱动程序
Class.forName("com.mysql.jdbc.Driver");
//注册 MySQL 驱动程序
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
//指定连接数据库的 URL、用户名和口令
String dbUrl = "jdbc:mysql://localhost:3306/SAMPLEDB";
String dbUser="root";
String dbPwd="1234";
//建立数据库连接
Connection con = java.sql.DriverManager.getConnection(dbUrl,
    dbUser,dbPwd);
```

Connection 类提供了以下用于控制事务的方法。

- **setAutoCommit(boolean autoCommit):** 设置是否自动提交事务。
- **commit():** 提交事务。
- **rollback():** 撤销事务。

对于新建的 Connection 实例，在默认情况下采用自动提交事务模式。可以通过 **setAutoCommit(false)** 方法来设置手工提交事务模式，然后就可以把多条更新数据库的 SQL 语句作为一个事务，在所有操作完成后调用 **commit()** 方法来整体提交事务，倘若其中一项 SQL 操作失败，就会抛出相应的 **SQLException**，此时应该在捕获异常的代码块中调用 **rollback()** 方法撤销事务。示例如下：

```
try {
    con = java.sql.DriverManager.getConnection(dbUrl,dbUser,dbPwd);
    //设置手工提交事务模式
    con.setAutoCommit(false);
    stmt = con.createStatement();
    //数据库更新操作 1
    stmt.executeUpdate("update ACCOUNTS set BALANCE=900 where ID=1 ");
    //数据库更新操作 2
    stmt.executeUpdate("update ACCOUNTS set BALANCE=1000 where ID=2 ");
    con.commit(); //提交事务
} catch (Exception e) {
    try {
        con.rollback(); //操作不成功则撤销事务
    } catch (Exception ex) {
        //处理异常
        .....
    }
    //处理异常
    .....
} finally {
    try {
        stmt.close();
        con.close();
    } catch (Exception ex) {
        //处理异常
        .....
    }
}
```

对于基于 Hibernate 的 Java 应用，虽然直接通过 JDBC API 来声明事务是可行的，但不提倡这种声明事务的方式，因为它使得 Java 应用既要和 Hibernate API 绑定在一起，又要和 JDBC API 绑定在一起，这削弱了 Java 应用的独立性，不利于程序的维护和升级。

## 14.5 Java应用通过Hibernate API声明事务

Hibernate 封装了 JDBC API 和 JTA API, Java 应用直接通过 Hibernate API 来声明事务, 有两个优点:

(1) 有利于跨平台开发。

(2) 当应用程序通过 Hibernate API 声明事务时, 事务与 Session 更加紧密地集成在一起。例如, 当调用 `org.hibernate.Transaction` 的 `commit()` 方法来提交事务时, 该方法在默认情况下 (清理缓存模式为 `FlushMode.AUTO`) 会自动清理 Session 的缓存。

下面介绍在程序中声明事务的过程。

首先必须获得一个 Session 实例:

```
Session session=sessionFactory.openSession();
```

在 Hibernate API 中, Session 和 Transaction 接口提供了以下声明事务边界的方法。

(1) 声明事务的开始边界:

```
Transaction tx=session.beginTransaction();
```

以上方法完成两个任务:

- 为 Session 对象分配数据库连接, 并且自动把这个连接设为手工提交事务模式。Hibernate 的底层实现会自动调用代表数据库连接的 `java.sql.Connection` 对象的 `setAutoCommit(false)` 方法。
- 开始一个新的事务。

(2) 提交事务:

```
tx.commit();
```

以上方法完成以下任务:

- 在默认情况下, Session 采用自动清理缓存模式, 在这种模式下, `commit()` 方法会先自动调用 Session 的 `flush()` 方法清理缓存, 即按照 Session 缓存中对象的变化去同步更新数据库。
- 向底层数据库提交事务。
- 释放 Session 占用的数据库连接。

### Tips

Session 的 `flush()` 方法根据缓存中的持久化对象的状态变化来同步更新数据库。`flush()` 方法会执行一系列的 `insert`、`update` 和 `delete` 语句, 但是 `flush()` 方法不会提交事务。



## (3) 撤销事务:

```
tx.rollback();
```

以上方法立即撤销事务，并且释放 Session 占用的数据库连接。

从 Hibernate 3 版本开始，当调用 Session.beginTransaction()方法开始一个新事务时，Session 会自动从数据库连接池中获得一个新的连接。当调用 Transaction.commit()方法提交事务时，Session 会自动释放当前的数据库连接。由此可见，在 Hibernate 自身的实现中，从提高程序性能的角度出发，只有当 Session 与一个事务绑定时，才会占用数据库连接，这可以尽量缩短 Session 占用数据库资源的时间。

### 14.5.1 处理异常

在 Hibernate 3 以前的版本中，Hibernate 抛出的异常都属于受检查异常(Checked Exception)，因此程序必须捕获并处理这种异常。Hibernate 之所以抛出受检查异常，主要是受 JDBC API 的影响，因为 JDBC API 抛出的异常都属于受检查异常。当 Hibernate 被广泛运用到实际 Java 应用中后，人们发现，在多数情况下，Hibernate 抛出的异常都是致命的，开发人员处理这种异常的最好方式是：释放程序占用的资源，向用户端显示错误消息，然后就终止应用程序。基于上述原因，从 Hibernate 3 版本开始，Hibernate 抛出的异常都是运行时异常。例程 14-1 演示了处理 Hibernate 异常的基本流程。

例程 14-1 处理 Hibernate 异常的基本流程

```
public void doWork(){
    Session session = factory.openSession();
    Transaction tx;
    try {
        tx = session.beginTransaction(); //开始一个事务
        //执行一些操作
        ...
        tx.commit(); //提交事务
    }catch (RuntimeException e) {
        if (tx!=null){
            try{
                tx.rollback(); //操作不成功则撤销事务
            }catch(RuntimeException ex){
                log.error("无法撤销事务",ex); //记录日志
            }
        }
        throw e; //继续抛出异常
    }finally {
        try{
```

```

        session.close();
    }catch(RuntimeException ex){
        log.error("无法关闭 Session",ex); //记录日志
    }
}
}

```

以上 `doWork()` 方法用于执行一个事务，如果执行事务时出现异常，就在 `catch` 代码块中撤销事务，然后继续抛出异常。不管事务成功与否，最后都应该调用 `Session` 的 `close()` 方法来关闭 `Session`，所以通常在 `finally` 代码块中关闭 `Session`。

当 `doWork()` 方法继续抛出异常后，对于调用 `doWork()` 方法的高层客户程序，如何处理这个异常呢？由于 `Hibernate` 抛出的异常通常都是致命的，因此高层客户程序的一般处理方式为：释放程序占用的资源，向用户端显示错误消息，然后就终止应用程序。例外情况是 `doWork()` 方法抛出 `StaleObjectStateException`，这种异常是为了避免并发问题而出现的异常，如果出现这种异常，一种处理方式为不必终止应用程序，而是与用户端经过一番交互后，在一个新的 `Session` 对象中重新开始一个事务，本书第 15 章的 15.5 节将进一步介绍处理 `StaleObjectStateException` 的方式。

### 14.5.2 Session与事务的关系

关于 `Session` 与事务的关系，有以下值得注意的地方。

#### 1. 及时提交或撤销事务

即使事务中仅包含只读操作，也应该在事务执行成功后提交事务，并且在事务执行失败时撤销事务，因为在提交或撤销事务时，数据库系统会释放事务所占用的资源，这有利于提高数据库的运行性能。

#### 2. 一个Session对应多个事务

在本章 14.5.1 节的例程 14-1 的 `doWork()` 方法中，一个 `Session` 对象仅用来执行一个事务，事务提交后，就关闭了这个 `Session` 对象。实际上，一个 `Session` 也可以对应多个事务，例如：

```

try{
    tx1 = session.beginTransaction(); //开始第一个事务
    //执行一些操作，加载 Account 对象
    Account account=(Account)a.get(Account.class,new Long(1));
    ...
    tx1.commit(); //提交第一个事务

    //执行一些耗时的操作，这段操作不属于任何数据库事务
    double amount=new Scanner(System.in).nextDouble();
}

```

```
//等待用户输入取款数额
account.setBalance(account.getBalance()-amount);
//修改 Account 对象的属性

tx2= session.beginTransaction(); //开始第二个事务
//执行一些操作
...
tx2.commit(); //提交第二个事务

}catch(RuntimeException e){
    //撤销事务
    if(tx1!=null)tx1.rollback();
    if(tx2!=null)tx2.rollback();
    throw e;
}finally{
    //关闭 Session
    session.close();
}
```

用一个 Session 对象来执行多个相关的数据库事务的优点在于：这些事务能够重用 Session 缓存中的持久化对象，避免多个相关的数据库事务重复到数据库加载相同的数据。在以上程序代码中，当第一个事务提交后，Session 不再占用数据库连接，但此时 account 对象仍然处于 Session 对象的缓存中。接下来可以执行一些耗时的操作，这段操作不属于任何数据库事务：先等待用户输入取款数额，然后修改 account 对象的 balance 属性。接下来开始第二个事务时，Session 再次获得数据库连接，当提交第二个事务时，Hibernate 会自动根据 account 对象的状态变化来同步更新数据库。

值得注意的是，在任何时候，一个 Session 只允许有一个未提交的事务。以下代码对一个 Session 同时声明了两个未提交的事务，这是不允许的：

```
tx1 = session.beginTransaction(); //开始第一个事务
tx2= session.beginTransaction(); //开始第二个事务
//执行一些操作
.....
tx1.commit(); //提交第一个事务
tx2.commit(); //提交第二个事务
```

### 3. 放弃使用出现异常的Session

如果在执行 Session 的一个事务时出现了异常，就必须立即关闭这个 Session，不能再利用这个 Session 来执行其他的事务。因为一旦执行 Session 的一个事务时出现了异常，那么 Session 的缓存中可能会出现不一致的数据，所以不能再使用这个 Session。例如，以下代码把两个事务各自放在单独的 try-catch 代码块中，即使第一

个事务执行失败，仍然会执行执行第二个事务，这种做法是不可取的：

```
try{
    try{
        tx1 = session.beginTransaction(); //开始第一个事务
        //执行一些操作
        .....
        tx1.commit(); //提交第一个事务
    }catch(RuntimeException e){
        if(tx1!=null)tx1.rollback();
    }

    try{
        tx2= session.beginTransaction(); //开始第二个事务
        //执行一些操作
        .....
        tx2.commit(); //提交第二个事务
    }catch(RuntimeException e){
        if(tx2!=null)tx2.rollback();
    }

}finally{
    //关闭 Session
    session.close();
}
```

#### 4. 自动提交事务模式

Hibernate 配置文件中有一个 `hibernate.connection.autocommit` 属性，它的默认值为 `false`。如果把它设为 `true`，那么 Hibernate 为 Session 对象分配的数据库连接将采用自动提交事务模式。在这种情况下，可以不用在 Session 中显式声明事务边界。以下程序代码试图保存两个 Account 对象：

```
Session session=sessionFactory.openSession();
Long generatedId1=session.save(account1);
Long generatedId2=session.save(account2);
session.flush(); //清理 Session 缓存
session.close();
```

在自动提交事务模式下，当清理 Session 缓存时，会执行两个 SQL insert 语句，每一个 insert 语句都对应一个独立的事务。

由于在自动提交事务模式下，无法把多个 SQL 语句放到同一个事务中，因此在实际应用中很少使用这种事务模式。

### 14.5.3 设定事务超时

`org.hibernate.Transaction` 接口的 `setTimeout(int seconds)` 方法用于设定事务超时的时间，以秒为单位。例如，以下代码把事务超时时间设为 5 秒：

```
Transaction tx = session.beginTransaction(); //开始一个事务
tx.setTimeout(5); //设定事务超时时间
//执行一些操作
...
tx.commit(); //提交第一个事务
```

值得注意的是，底层 JDBC 驱动程序实际上并没有能力监控事务是否超时，而只能监控 `PreparedStatement` 执行 SQL 语句是否超时，如果超时，就抛出 `SQLException`。

## 14.6 小结

数据库事务由一组在业务逻辑上相互依赖的 SQL 语句组成，它必须具备 ACID 特征。数据库管理系统采用日志来保证事务的原子性、一致性和持久化性，采用锁机制来实现事务的隔离性。

Hibernate 封装了 JDBC API 和 JTA API。以下程序展示了通过 Hibernate API 来声明事务的正常流程：

```
Session session = sessionFactory.openSession();
Transaction tx= session.beginTransaction(); //声明开始一个事务
//执行一些操作
...
tx.commit(); //提交事务
```

## 第 15 章 处理并发问题

第 14 章已经介绍了数据库事务的概念，以及在 Java 程序中声明事务边界的方法。第 14 章仅考虑了运行单个事务的情况。而在实际应用中，往往会有多个用户同时访问数据库系统，分别执行各自的事务。在这样的并发环境中，当多个事务同时访问相同的数据资源时，可能会造成各种并发问题。可以通过设定数据库系统的事务隔离级别来避免各类并发问题，此外，在应用程序中还可以采用悲观锁和乐观锁来解决丢失更新这一并发问题。本章将介绍事务隔离级别、悲观锁和乐观锁等概念，并且介绍在应用程序中设置事务隔离级别及运用悲观锁和乐观锁的方法。

### 15.1 多个事务并发运行时的并发问题

在并发环境中，一个数据库系统会同时为各种各样的客户程序提供服务，这些客户程序可以是 `mysql.exe` 客户程序，也可以是 Java 应用程序，有的 Java 应用程序在运行时可能还包含多个线程。图 15-1 列出了某一时刻多个客户程序同时访问数据库系统的状态。

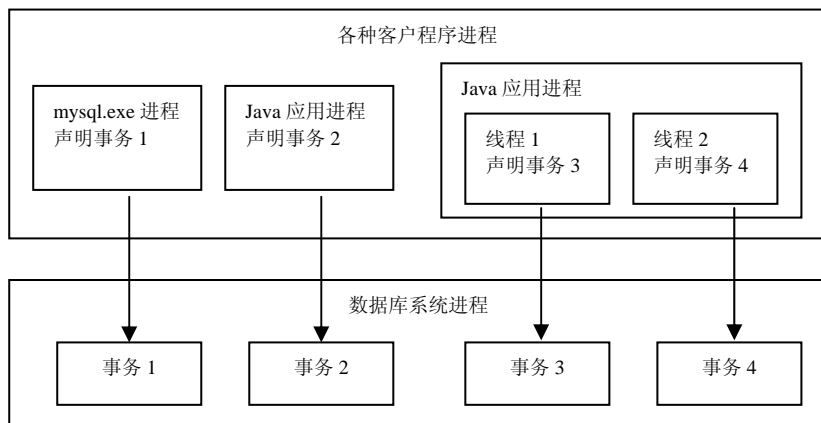


图 15-1 在并发环境中某个时刻各种客户程序同时访问数据库系统

对于同时运行的多个事务，当这些事务访问数据库中相同的数据时，如果没有采取必要的隔离机制，就会导致各种并发问题，这些并发问题可归纳为以下几类。

- 第一类丢失更新：撤销一个事务时，把其他事务已提交的更新数据覆盖。
- 脏读：一个事务读到另一事务未提交的更新数据。
- 虚读：一个事务读到另一事务已提交的新插入的数据。
- 不可重复读：一个事务读到另一事务已提交的更新数据。

- 第二类丢失更新：这是不可重复读中的特例，一个事务覆盖另一事务已提交的更新数据。

例如，花果山决定在网上进行公开投票，选举出一名值得信赖的猴子，来管理花果山的桃园。假定猴子甲和猴子乙都要给智多星投一票。猴子甲先投票，这个投票事务包括以下操作：

- (1) 查询智多星当前的票数，为 1 000。
- (2) 给智多星增加 1 票，智多星的票数变为 1 001。

猴子乙接着投票，这个投票事务包括以下操作：

- (1) 查询智多星当前的票数，为 1 001。
- (2) 给智多星增加 1 票，智多星的票数变为 1 002。

由此可见，如果以上两个投票事务在时间上错开运行，不会有任何问题，但是如果猴子甲和猴子乙同时给智多星投票，就可能出现以上 5 种并发问题，下面分别介绍。

### 15.1.1 第一类丢失更新

这种并发问题是由于完全没有隔离事务造成的。当两个事务更新相同的数据资源，如果一个事务被提交，另一个事务却被撤销，那么会连同第一个事务所做的更新也被撤销。如表 15-1 所示，假如猴子乙投票事务在 T6 时刻被提交，那么智多星的票数变为 1 001。在 T8 时刻，猴子甲改变想法，撤销对智多星的投票，猴子甲的投票事务被撤销，智多星的票数退回到执行该事务前的初始状态，因此票数又恢复为 1 000。由于猴子乙投票事务对票数所做的更新被覆盖，智多星损失了猴子乙所投的 1 票。

表 15-1 并发运行的两个事务导致第一类丢失更新

时 间	猴子甲投票事务	猴子乙投票事务
T1	开始事务	
T2		开始事务
T3	查询智多星的票数为 1 000	
T4		查询智多星的票数为 1 000
T5		把票数改为 1 001
T6		提交事务
T7	把票数改为 1 001	
T8	撤销事务，智多星的票数恢复为 1 000	

## Tips

既然两个事务同时运行,为什么表 15-1 中的每一步操作都发生在不同的时刻?这是因为数据库服务器在某个确定的时刻只可能执行一条 SQL 语句,可以把表 15-1 中的 T3 和 T4 理解为精确到毫秒或微秒的时间,假如 T3 代表 10 点 10 分 10 秒 100 毫秒,而 T4 代表 10 点 10 分 10 秒 101 毫秒,那么从宏观上看,可以认为在这两个时刻执行的操作是并发的,也可以说是同时进行的。

## 15.1.2 脏读

如果第二个事务查询到了第一个事务未提交的更新数据,第二个事务依据这个查询结果继续执行相关的操作,但是接着第一个事务撤销了所做的更新,这会导致第二个事务操纵脏数据。如表 15-2 所示,猴子甲投票事务在 T5 时刻把票数改为 1 001,猴子乙投票事务在 T6 时刻查询智多星的票数为 1 001,猴子甲投票事务在 T7 时刻被撤销,猴子乙投票事务在 T8 时刻把票数改为 1 002。

由于猴子乙投票事务查询到了猴子甲投票事务未提交的更新数据,并且在这个查询结果的基础上进行更新操作,结果虽然猴子甲投票事务最后被撤销,智多星还是得到了猴子甲所投的 1 票。

表 15-2 并发运行的两个事务导致脏读

时 间	猴子甲投票事务	猴子乙投票事务
T1	开始事务	
T2		开始事务
T3	查询智多星的票数为 1 000	
T4		
T5	把票数改为 1 001	
T6		查询智多星的票数为 1 001 (脏读)
T7	撤销该事务,把票数恢复为 1 000	
T8		把票数改为 1 002
T9		提交事务

## 15.1.3 虚读

虚读是由于一个事务查询到了另一事务已提交的新插入的数据引起的。如表 15-3 所示,假定一个网站的统计事务在两个时刻统计所有注册客户的总数,在这两个时刻中间一个注册事务新注册了一个客户,那么就会导致统计事务两次统计结果不一样。统计事务无法相信查询结果,因为查询结果是不确定的,随时可能被其他事务改变。



表 15-3 并发运行的两个事务导致虚读

时 间	注 册 事 务	统 计 事 务
T1	开始事务	
T2		开始事务
T3		统计网站的注册客户的总数为 10 000 人
T4	注册一个新客户	
T5	提交事务	
T6		统计网站的注册客户的总数为 10 001 人（虚读）
T7		到底是哪个统计数据有效？

对于实际应用，在一个事务中不会对相同的数据查询两次，假定统计事务在 T3 时刻统计注册客户总数，执行的 select 语句为：

```
select count(*) from CUSTOMERS;
```

在 T6 时刻不再查询数据库，而是直接打印出统计结果为 10 000，这个统计结果与数据库中的当前数据有出入，确切地说，它反映的是 T3 时刻的数据状态，而不是当前的数据状态。

应该根据实际需要来决定是否允许虚读。以上面的统计事务为例，如果仅想大致了解一下注册客户总数，那么可以允许虚读；如果在同一个事务中，会依据查询的结果做出精确的决策，那么就必须采取必要的事务隔离措施，避免虚读。

### 15.1.4 不可重复读

不可重复读是由于一个事务查询到了另一事务已提交的对数据的更新引起的。当第二个事务在某一时刻查询某条记录，在另一时刻再查询相同记录时，看到了第一个事务已提交的对这条记录的更新，第二个事务无法判断到底以哪一时刻查询到的记录作为计算的基础，因为任何时候查询到的数据都有可能立刻被其他事务更新。

如表 15-4 所示，假如猴子乙投票事务两次查询智多星的票数，但得到了不同的查询结果，这使得猴子乙无法相信查询结果，因为查询结果是不确定的，随时可能被其他事务改变。

表 15-4 并发运行的两个事务导致不可重复读

时 间	猴子甲投票事务	猴子乙投票事务
T1	开始事务	
T2		开始事务
T3	查询智多星的票数为 1 000	
T4		查询智多星的票数为 1 000
T5	把票数改为 1 001	

(续表)

时 间	猴子甲投票事务	猴子乙投票事务
T6	提交事务	
T7		查询智多星的票数为 1 001
T8		到底是把票数改为 1 000+1, 还是 1 001+1

15.1.5 第二类丢失更新

第二类丢失更新是在实际应用中经常遇到的并发问题，它和不可重复读本质上是同一类并发问题，通常把它看做是不可重复读的一个特例。当两个或多个事务查询同样的记录，然后各自基于最初查询的结果更新该记录时，会造成第二类丢失更新问题。每个事务都不知道其他事务的存在，最后一个事务对记录所做的更新将覆盖由其他事务对该记录所做的已提交的更新。

如表 15-5 所示，猴子甲投票事务在 T5 时刻根据在 T3 时刻的查询结果，把票数改为 1 001，在 T6 时刻提交事务。猴子乙投票事务在 T7 时刻根据在 T4 时刻的查询结果，把票数改为 1 001。由于猴子乙投票事务覆盖了猴子甲投票事务对票数所做的更新，导致智多星损失了猴子甲所投的 1 票。

表 15-5 并发运行的两个事务导致第二类丢失更新		
时 间	猴子甲投票事务	猴子乙投票事务
T1	开始事务	
T2		开始事务
T3	查询智多星的票数为 1 000	
T4		查询智多星的票数为 1 000
T5	把票数改为 1 001	
T6	提交事务	
T7		把票数改为 1 001
T8		提交事务

15.2 数据库系统的锁的基本原理

在数据库系统的 ACID 特性中，隔离性就是指数据库系统必须具有隔离并发运行的各个事务的能力，使它们不会相互影响，避免出现 15.1 节介绍的各种并发问题，以保证数据的完整性和一致性。数据库系统采用锁来实现事务的隔离性。各种大型数据库采用的锁的基本理论是一致的，但在具体实现上各有差别。锁的基本原理如下：

- 当一个事务访问某种数据库资源时，如果执行 select 语句，必须先获得共

享锁，如果执行 insert、update 或 delete 语句，必须获得独占锁，这些锁用于锁定被操纵的资源。

- 当第二个事务也要访问相同的资源时，如果执行 select 语句，也必须先获得共享锁，如果执行 insert、update 或 delete 语句，也必须获得独占锁。此时根据已经放置在资源上的锁的类型，来决定第二个事务应该等待第一个事务解除对资源的锁定，还是可以立刻获得锁，表 15-6 列出了不同情况下第二个事务的进展。

表 15-6 根据已放置在资源上的锁来决定第二个事务能否立刻获得特定类别的锁

资源上已经放置的锁	第二个事务进行读操作	第二个事务进行更新操作
无	立即获得共享锁	立即获得独占锁
共享锁	立即获得共享锁	等待第一个事务解除共享锁
独占锁	等待第一个事务解除独占锁	等待第一个事务解除独占锁

许多数据库系统都有自动管理锁的功能，它们能根据事务执行的 SQL 语句，自动在保证事务间的隔离性与保证事务间的并发性能之间做出权衡，然后自动为数据库资源加上适当的锁，在运行期间还会自动升级锁的类型，以优化系统的性能。事务间的并发性能是指数据库系统能够同时执行多个事务，很少出现因为一个事务占用了特定资源，而导致其他事务必须暂停下来长时间等待资源的情况。

对于普通的并发性事务，通过数据库系统的自动锁定管理机制基本可以保证事务之间的隔离性，但如果对数据安全、数据库完整性和一致性有特殊要求，也可以由事务本身来控制对数据资源的锁定和解锁，本章 15.4 节、15.5 节和 15.6 节会对此做进一步介绍。

## 15.3 数据库的事务隔离级别

尽管数据库系统允许用户在事务中显式地为数据资源加锁(参见本章 15.4 节)，但是首先应该考虑让数据库系统自动管理锁，它会分析事务中的 SQL 语句，然后自动为 SQL 语句所操纵的数据资源加上合适的锁，而且在锁的数目太多时，数据库系统会自动进行锁升级，以提高系统性能。

锁机制能有效地解决各种并发问题，但是它会影响并发性能。并发性能越好，数据库系统同时为各种客户程序提供服务的能力就越强。当一个事务锁定数据资源时，其他事务必须停下来等待，这就降低了数据库系统同时响应各种客户程序的速度。

为了能让用户根据实际应用的需要，在事务的隔离性与并发性之间做出合理的权衡，数据库系统提供了 4 种事务隔离级别供用户选择。

- **Serializable**: 串行化。
- **Repeatable Read**: 可重复读。
- **Read Committed**: 读已提交数据。
- **Read Uncommitted**: 读未提交数据。

数据库系统采用不同的锁类型来实现以上 4 种隔离级别，具体的实现过程对用户是透明的。用户应该关心的是如何选择合适的隔离级别。在 4 种隔离级别中，**Serializable** 的隔离级别最高，**Read Uncommitted** 的隔离级别最差，表 15-7 列出了各种隔离级别所能避免的并发问题。

表 15-7 各种隔离级别所能避免的并发问题

隔 离 级 别	是否出现第一类 丢失更新	是否出现脏读	是否出现虚读	是否出现不可 重复读	是否出现第二类 丢失更新
Serializable	否	否	否	否	否
Repeatable Read	否	否	是	否	否
Read Committed	否	否	是	是	是
Read Uncommitted	否	是	是	是	是

1. **Serializable（串行化）**

当数据库系统使用 **Serializable** 隔离级别时，一个事务在执行过程中完全看不到其他事务对数据库所做的更新。当两个事务同时操纵数据库中相同数据时，如果第一个事务已经在访问该数据，第二个事务只能停下来等待，必须等到第一个事务结束后才能恢复运行。因此这两个事务实际上以串行化方式运行。

2. **Repeatable Read（可重复读）**

当数据库系统使用 **Repeatable Read** 隔离级别时，一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，但是不能看到其他事务对已有记录的更新。

3. **Read Committed（读已提交数据）**

当数据库系统使用 **Read Committed** 隔离级别时，一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，而且能看到其他事务已经提交的对已有记录的更新。

4. **Read Uncommitted（读未提交数据）**

当数据库系统使用 **Read Uncommitted** 隔离级别时，一个事务在执行过程中可以看到其他事务没有提交的新插入的记录，而且能看到其他事务没有提交的对已有记录的更新。

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大，图 15-2 显示了隔离级别与并发性能的关系。对于多数应用程序，可以优先考

虑把数据库系统的隔离级别设为 **Read Committed**，它能够避免脏读，而且具有较好的并发性能。尽管它会导致不可重复读、虚读和第二类丢失更新这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制，参见本章 15.4 节、15.5 节和 15.6 节。

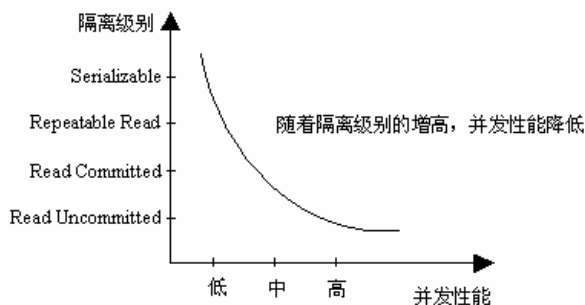


图 15-2 隔离级别与并发性能的关系

### 15.3.1 在mysql.exe程序中设置隔离级别

每启动一个 **mysql.exe** 程序，就会获得一个单独的数据库连接。每个数据库连接都有一个全局变量 **@@tx\_isolation**，表示当前的事务隔离级别。**MySQL** 默认的隔离级别为 **Repeatable Read**。如果要查看当前的隔离级别，可使用如下 **SQL** 命令：

```
mysql> select @@tx_isolation;
```

```
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+
```

如果要把当前 **mysql.exe** 程序的隔离级别改为 **Read Committed**，可使用如下 **SQL** 命令：

```
mysql> set transaction isolation level read committed;
```

如果要设置数据库系统的全局的隔离级别，可使用如下 **SQL** 命令：

```
mysql> set global transaction isolation level read committed;
```

### 15.3.2 在应用程序中设置隔离级别

**JDBC** 数据库连接使用数据库系统默认的隔离级别。在 **Hibernate** 的配置文件中可以显式地设置隔离级别。每一种隔离级别都对应一个整数：

- 1: Read Uncommitted

- 2: Read Committed
- 4: Repeatable Read
- 8: Serializable

例如，以下代码把 `hibernate.properties` 配置文件中的隔离级别设为 `Read Committed`：

```
hibernate.connection.isolation=2
```

对于从数据库连接池中获得的每一个连接，`Hibernate` 都会把它改为使用 `Read Committed` 隔离级别。

## 15.4 在应用程序中采用悲观锁

当数据库系统采用 `Read Committed` 隔离级别时，会导致不可重复读和第二类丢失更新的并发问题。在可能出现这种问题的场合，可以在应用程序中采用悲观锁或乐观锁来避免这类问题。从应用程序的角度，锁可以分为以下几类。

- 悲观锁：指在应用程序中显式地为数据资源加锁。悲观锁假定当前事务操纵数据资源时，肯定还会有其他事务同时访问该数据资源，为了避免当前事务的操作受到干扰，先锁定资源。尽管悲观锁能够防止丢失更新和不可重复读这类并发问题，但是它会影响并发性能，因此应该很谨慎地使用悲观锁。
- 乐观锁：乐观锁假定当前事务操纵数据资源时，不会有其他事务同时访问该数据资源，因此完全依靠数据库的隔离级别来自动管理锁的工作。应用程序采用版本控制手段来避免可能出现的并发问题。

### Tips

从这两种锁的实现机制可以看出，悲观锁对事态估计很悲观，总是认为其他事务会占用待访问的资源；而乐观锁对事态估计很乐观，先不考虑其他事务是否会占用待访问的资源。悲观锁与乐观锁由此得名。

悲观锁有两种实现方式。

- 方式一：在应用程序中显式指定采用数据库系统的独占锁来锁定数据资源。
- 方式二：在数据库表中增加一个表明记录状态的 `LOCK` 字段，当它取值为“Y”时，表示该记录已经被某个事务锁定，如果为“N”，表明该记录处于空闲状态，事务可以访问它。

本书介绍第一种实现方式。当一个事务执行 `select` 语句时，在默认情况下，数据库系统会采用共享锁来锁定查询的记录。此外，`MySQL`、`Oracle` 和 `Ms SQL` 都支持以下形式的 `select` 语句：

```
select ... for update
```

以上语句显式指定采用独占锁来锁定查询的记录。执行该查询语句的事务持有这把锁，直到事务结束才会释放锁。在执行事务过程中，其他事务如果要查询、更新或删除这些被锁定的记录，必须等到第一个事务执行结束，才能有机会操纵这些记录。

例如，对于并发运行的猴子甲投票事务和猴子乙投票事务，假定猴子甲投票事务先执行以下语句：

```
select * from MONKEYS where ID=1 for update;
```

那么这条 ID 为 1 的 MONKEYS 记录就被锁定。其他事务如果也要对这条记录进行查询、更新或删除操作，就必须停下来等待，直到猴子甲投票事务结束，其他事务才有机会访问这条记录。表 15-8 列出了猴子甲投票事务和猴子乙投票事务的执行过程。

表 15-8 利用悲观锁协调并发运行的猴子甲投票事务和猴子乙投票事务

时 间	猴子甲投票事务	猴子乙投票事务
T1	开始事务	
T2		开始事务
T3	select * from MONKEYS where ID=1 for update; 查询结果显示票数为 1 000; 这条记录被锁定	
T4		select * from MONKEYS where ID=1 for update; 执行该语句时，事务停下来等待猴子甲投票事务解除对这条记录的锁定
T5	把票数改为 1 001	
T6	提交事务	
T7		事务恢复运行，查询结果显示票数为 1 001。这条记录被锁定
T8		把票数改为 1 002
T9		提交事务

在 Hibernate 应用中，当通过 Session 的 get()和 load()方法来加载一个对象时，可以采用以下方式声明使用悲观锁：

```
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1),
    LockMode.UPGRADE);
```

org.hibernate.LockMode 类表示锁定模式，表 15-9 列出了它的几个静态实例的作用。

表 15-9 LockMode 类表示的几种锁定模式

锁定模式	描 述
LockMode.NONE	如果在 Hibernate 的缓存中存在 Monkey 对象,就直接返回该对象的引用;否则就通过 select 语句到数据库中加载该对象。这是默认值
LockMode.READ	不管 Hibernate 的缓存中是否存在 Monkey 对象,总是通过 select 语句到数据库中加载该对象;如果映射文件中设置了版本元素,就执行版本检查,比较缓存中的 Monkey 对象是否和数据库中 Monkey 对象的版本一致
LockMode.UPGRADE	不管 Hibernate 的缓存中是否存在 Monkey 对象,总是通过 select 语句到数据库中加载该对象;如果映射文件中设置了版本元素,就执行版本检查,比较缓存中的 Monkey 对象是否和数据库中 Monkey 对象的版本一致;如果数据库系统支持悲观锁(如 Oracle 和 MySQL),就执行 select ... for update 语句,如果数据库系统不支持悲观锁(如 Sybase),就执行普通的 select 语句
LockMode.UPGRADE_NOWAIT	和 LockMode.UPGRADE 具有同样的功能。此外,对于 Oracle 等支持“update nowait”的数据库,执行 select...for update nowait 语句,“nowait”表明如果执行该 select 语句的事务不能立刻获得悲观锁,那么不会等待其他事务释放锁,而是立刻抛出一个锁定异常
LockMode.WRITE	当 Hibernate 向数据库保存一个对象时,会自动使用这种锁定模式。这种锁定模式仅供 Hibernate 内部使用,在应用程序中不应该使用它

在默认情况下, Session 的 get()和 load()方法的锁定模式为 LockMode.NONE,如果设为 LockMode.UPGRADE,就表示采用悲观锁。对于 Oracle 数据库,还可以用 LockMode.UPGRADE\_NOWAIT 来表明使用悲观锁。

LockMode.READ 主要和 Session 的 lock()方法联合使用,用于对一个游离对象进行版本检查,本章 15.5.3 节会对此做详细介绍。

下面通过具体的例子介绍悲观锁的运行机制。本节的范例程序位于配套光盘的 sourcecode\chapter15\15.4 目录下。运行该程序之前,需要先在 SAMPLEDB 数据库中手工创建 MONKEYS 表,然后加入测试数据,相关的 SQL 脚本文件为 15.4\schema\sampledb.sql。

MONKEYS 表的 DDL 定义如下,其中 COUNT 字段表示猴子所得的票数:

```
create table MONKEYS(
    ID bigint not null,
    NAME varchar(15),
    COUNT int,
    primary key (ID)
) type=INNODB;
```

在范例程序的 hibernate.properties 文件中,把数据库的事务隔离级别设为 Read Committed 级别:

```
hibernate.connection.isolation=2
```



在 chapter15 目录下有两个 ANT 的工程文件：`build1.xml` 和 `build2.xml`，它们的区别在于文件开头设置的路径不一样，例如在 `build1.xml` 文件中设置了以下路径：

```
<property name="source.root" value="15.4/src"/>
<property name="class.root" value="15.4/classes"/>
<property name="lib.dir" value="lib"/>
<property name="schema.dir" value="15.4/schema"/>
```

在 DOS 命令行下进入 chapter15 根目录，然后输入命令：

```
ant -file build1.xml run
```

就会运行 `BusinessService` 类。ANT 命令的 `-file` 选项用于显式指定工程文件。例程 15-1 是 `BusinessService` 类的源程序。

**例程 15-1** `BusinessService` 类

```
package mypack;

import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class BusinessService extends Thread{
    public static SessionFactory sessionFactory;
    static{.....} //初始化 Hibernate

    private Log log; //用于记录事务运行中的日志

    public BusinessService(String name,Log log){
        super(name);
        this.log=log;
    }

    public void run(){
        try{
            vote(); //执行投票事务
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public void vote()throws Exception{.....} //投票事务

    public static void main(String args[]) throws Exception {
        Log log=new Log();
        Thread thread1=new BusinessService("猴子甲投票事务",log);
        Thread thread2=new BusinessService("猴子乙投票事务",log);
```

```

        thread1.start(); //启动猴子甲投票事务
        thread2.start(); //启动猴子乙投票事务

        while(thread1.isAlive() || thread2.isAlive()){
            Thread.sleep(100);
        }
        log.print();
        sessionFactory.close();
    }
}

class Log{ /** 日志类 */
    private ArrayList logs=new ArrayList();

    synchronized void write(String text){
        logs.add(text);
    }
    public void print(){
        for (Iterator it = logs.iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
    }
}

```

**BusinessService** 类继承了 **java.lang.Thread** 类, 因此它是一个线程类。在 **main()** 方法中启动了两个 **BusinessService** 线程: **thread1** 和 **thread2** 线程, 它们都运行 **vote()** 方法。**thread1** 执行猴子甲投票事务, **thread2** 执行猴子乙投票事务。**vote()** 方法的代码如下:

```

        tx = session.beginTransaction();
        log.write(getName()+" :开始事务");
        Thread.sleep(500);

        Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
        log.write(getName()+" :查询到智多星的票数为"+monkey.getCount());
        Thread.sleep(500);

        monkey.setCount(monkey.getCount()+1);
        log.write(getName()+" :把智多星的票数改为"+monkey.getCount());

        log.write(getName()+" :提交事务");
        tx.commit();

        Thread.sleep(500);

```

为了便于演示这两个线程并发运行的效果，每个线程执行一些代码后就会睡眠片刻，把 CPU 让给另一个线程。为了跟踪这两个线程执行事务的时间顺序，`vote()`方法生成了一些日志，`main()`方法等到这两个线程结束后会输出这些日志。

当 `vote()`方法调用 `Session` 的 `get()`方法时，如果采用默认的 `LockMode.None` 模式，那么 `Hibernate` 执行的 `select` 语句为：

```
select * from MONKEYS where ID=1;
```

以上 `select` 语句表明应用程序没有使用悲观锁，当这两个线程并发运行时，最后生成的日志如下：

```
猴子甲投票事务:开始事务
猴子乙投票事务:开始事务
猴子乙投票事务:查询到智多星的票数为 1000
猴子甲投票事务:查询到智多星的票数为 1000
猴子乙投票事务:把智多星的票数改为 1001
猴子乙投票事务:提交事务
猴子甲投票事务:把智多星的票数改为 1001
猴子甲投票事务:提交事务
```

以上日志反映了这两个线程执行事务时可能会出现的一种随机执行顺序。从日志可以看出，猴子甲投票事务覆盖了猴子乙投票事务对票数所做的更新，导致智多星损失了猴子乙所投的 1 票。

由此可见，在数据库系统使用 `Read Committed` 隔离级别的情况下，如果应用程序没有采用悲观锁，当猴子甲投票事务和猴子乙投票事务并发运行时，会导致第二类丢失更新问题。

下面修改 `vote()`方法中的程序代码，使 `Session` 的 `get()`方法采用 `LockMode.UPGRADE` 模式：

```
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1),
    LockMode.UPGRADE);
```

在 `LockMode.UPGRADE` 模式下，当运行 `Session` 的 `get()`方法时，`Hibernate` 执行以下 `select` 语句：

```
select * from MONKEYS where ID=1 for update;
```

如果猴子甲投票事务和猴子乙投票事务同时执行以上 `select` 语句，只会有一个事务获得悲观锁，另一个事务必须等待，直到前一个事务结束，然后释放了锁，另一事务才能获得锁并恢复运行。应用程序最后输出以下日志：

```
猴子甲投票事务:开始事务
猴子乙投票事务:开始事务
猴子甲投票事务:查询到智多星的票数为 1000
猴子甲投票事务:把智多星的票数改为 1001
```

```

猴子甲投票事务:提交事务
猴子乙投票事务:查询到智多星的票数为 1001
猴子乙投票事务:把智多星的票数改为 1002
猴子乙投票事务:提交事务

```

从日志看出,不管猴子甲投票事务和猴子乙投票事务如何随机地并发运行,一个事务不会覆盖另一个事务对票数所做的更新。由此可见,使用悲观锁能有效地避免不可重复读和第二类丢失更新问题。但是,悲观锁会影响并发性能,导致一个事务锁定数据资源后,其他事务如果也要访问该资源,就必须先等待前一个事务执行结束。

## 15.5 利用Hibernate的版本控制来实现乐观锁

乐观锁是由应用程序提供的一种机制,这种机制既能保证多个事务并发访问数据,又能防止第二类丢失更新问题。在应用程序中,可以利用 **Hibernate** 提供的版本控制功能来实现乐观锁。对象-关系映射文件中的<version>元素和<timestamp>元素都具有版本控制功能。<version>元素利用一个递增的整数来跟踪数据库表中记录的版本,而<timestamp>元素用时间戳来跟踪数据库表中记录的版本。

### 15.5.1 使用<version>元素

下面介绍利用<version>元素对 **MONKEYS** 表中记录进行版本控制的步骤。

(1) 在 **Monkey** 类中定义一个代表版本信息的属性:

```

private int version;
public int getVersion() {
    return this.version;
}

public void setVersion(int version) {
    this.version = version;
}

```

(2) 在 **MONKEYS** 表中定义一个代表版本信息的字段:

```

create table MONKEYS (
    ID bigint not null,
    NAME varchar(15),
    COUNT int,
    VERSION integer,
    primary key (ID)
) type=INNODB;

```

(3) 在 Monkey.hbm.xml 文件中用<version>元素来建立 Monkey 类的 version 属性与 MONKEYS 表中 VERSION 字段的映射:

```
<id name="id" type="long" column="ID">
    <generator class="increment"/>
</id>
<version name="version" column="VERSION" />
.....
```

#### Tips

在映射文件中, <version>元素必须紧跟在<id>元素的后面。

(4) BusinessService 类与本章 15.4 节的例程 15-1 基本相同, 区别在于本节的 BusinessService 类的 vote()方法会捕获 StaleObjectStateException 异常。

```
public void vote()throws Exception{
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {

        tx = session.beginTransaction();
        log.write(getName()+":开始事务");
        Thread.sleep(500);

        Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
        log.write(getName()+":查询到智多星的票数为"+monkey.getCount());
        Thread.sleep(500);

        monkey.setCount(monkey.getCount()+1);
        log.write(getName()+":把智多星的票数改为"+monkey.getCount());
        Thread.sleep(500);

        tx.commit();
        log.write(getName()+":提交事务");
        Thread.sleep(500);

    }catch(StaleObjectStateException e){
        if (tx != null) {
            tx.rollback();
        }
        e.printStackTrace();
        System.out.println(getName()
            +":智多星票数已被其他事务修改,本事务被撤销,请重新开始投票事务");
        log.write(getName()+":智多星票数已被其他事务修改,本事务被撤销");
    }catch (RuntimeException e) {
        if (tx != null) {
            tx.rollback();
        }
    }
}
```

```

        throw e;
    }finally {
        session.close();
    }
}

```

接下来介绍加入版本控制后 Hibernate 的运行时行为。

当 Hibernate 加载一个 Monkey 对象时, 它的 version 属性表示 MONKEYS 表中相关记录的版本。当 Hibernate 更新一个 Monkey 对象时, 会根据 Session 缓存中 Monkey 对象的 id 与 version 属性的当前值到 MONKEYS 表中去定位匹配的记录, 假定 Session 缓存中 Monkey 对象的 version 属性为 0, 那么在猴子甲投票事务中 Hibernate 执行的 update 语句为:

```

update MONKEYS set NAME='智多星',COUNT=1001,VERSION=1
where ID=1 and VERSION=0;

```

如果存在匹配的记录, 就更新这条记录, 并且把 VERSION 字段的值增加为 1, 此外, 还会把 Session 缓存中 Monkey 对象的 version 属性也更新为 1。当猴子乙投票事务接着执行以下 update 语句时:

```

update MONKEYS set NAME='智多星',COUNT=1002,VERSION=1
where ID=1 and VERSION=0;

```

由于 ID 为 1 的 MONKEYS 记录的版本已经被猴子甲投票事务修改, 因此找不到匹配的记录, 此时 Hibernate 会抛出 StaleObjectStateException。

在应用程序中应该捕获该异常, 这种异常有两种处理方式。

- 方式一: 自动撤销事务, 通知用户得票信息已被其他事务修改, 需要重新开始事务。本例程就采用这种方式。
- 方式二: 通知用户得票信息已被其他事务修改, 显示最新票数, 由用户决定如何继续事务, 用户也可以决定立刻撤销事务。

本节范例程序位于配套光盘的 sourcecode\chapter15\15.5 目录下。在运行该程序之前, 需要先在 SAMPLEDB 数据库中手工创建 MONKEYS 表, 相关的 SQL 脚本文件为 15.5\schema\sampladb.sql。在 DOS 下转到根目录 chapter15, 输入命令:

```

ant -file build2.xml run

```

该命令将运行 BusinessService 类, 它最后输出如下日志:

```

猴子甲投票事务:开始事务
猴子乙投票事务:开始事务
猴子甲投票事务:查询到智多星的票数为 1000
猴子乙投票事务:查询到智多星的票数为 1000
猴子甲投票事务:把智多星的票数改为 1001
猴子甲投票事务:提交事务

```

猴子乙投票事务:把智多星的票数改为 1001  
猴子乙投票事务:智多星票数已被其他事务修改,本事务被撤销

表 15-10 列出了猴子甲投票事务和猴子乙投票事务并发运行的过程。

表 15-10 利用乐观锁协调并发的猴子甲投票事务和猴子乙投票事务		
时 间	猴子甲投票事务	猴子乙投票事务
T1	开始事务	
T2		开始事务
T3	select * from MONKEYS where ID=1; 查询结果显示票数为 1 000, 该记录的 VERSION 字段为 0	
T4		select * from MONKEYS where ID=1; 查询结果显示票数为 1 000, 该记录的 VERSION 字段为 0
T5	把票数改为 1 001。Hibernate 执行的 update 语句为: update MONKEYS set COUNT=1 001,VERSION=1 where ID=1 and VERSION=0;	
T6	提交事务	
T7		把票数改为 1 001。Hibernate 执行的 update 语句为: update MONKEYS set COUNT=1 001, VERSION=1 where ID=1 and VERSION=0; 没有找到匹配的记录, Hibernate 抛出 StaleObjectStateException
T8		应用程序撤销本事务, 通知用户得票信息已 被修改, 需要重新开始猴子乙投票事务

下面再通过一系列状态图来展示两个事务并发运行时 Session 缓存及数据库的状态变化。状态图中猴子甲投票事务简称为甲事务, 猴子乙投票事务简称为乙事务。

(1) 两个事务先后开始时, 两个 Session 缓存均为空, 数据库中 ID 为 1 的 MONKEYS 记录的 COUNT 字段为 1000, VERSION 字段为 0, 参见图 15-3。



图 15-3 两个事务先后开始时的状态

(2) 猴子甲投票事务及猴子乙投票事务先后执行以下程序代码：

```
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
```

以上程序代码通过 `Session.get()` 方法加载 `Monkey` 对象，这两个事务的 `Session` 缓存中均有一个 `OID` 为 1 的 `Monkey` 对象，它们的 `count` 属性为 1 000，`version` 属性为 0，参见图 15-4。

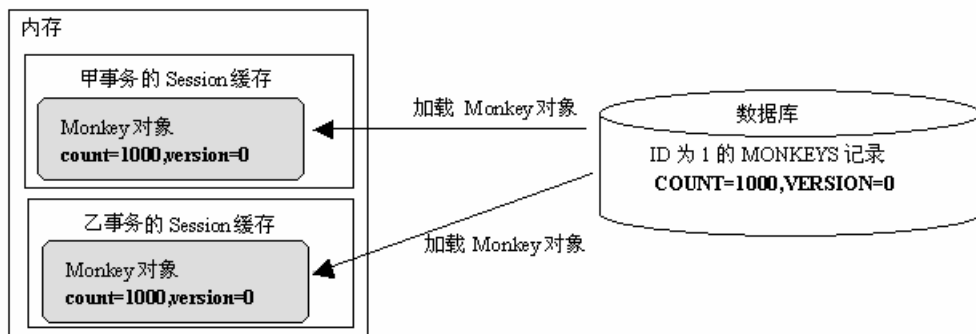


图 15-4 两个事务先后加载 `Monkey` 对象后的状态

(3) 猴子甲投票事务执行以下程序代码，修改 `Session` 缓存中 `Monkey` 对象的 `count` 属性，把它改为 1 001，参见图 15-5。

```
monkey.setCount(monkey.getCount()+1);
```

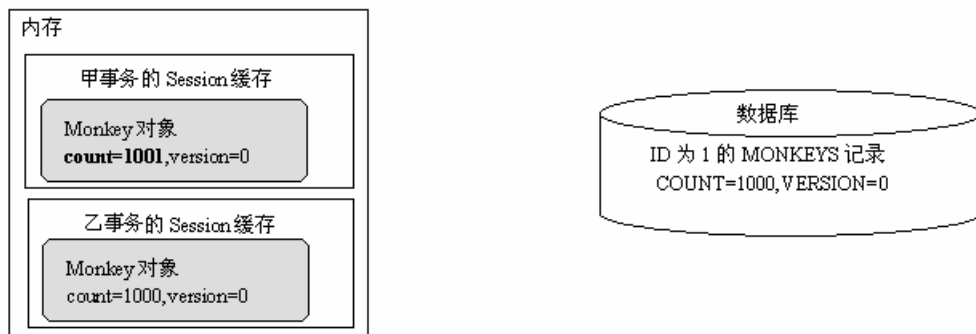


图 15-5 猴子甲投票事务修改 `Session` 缓存中 `Monkey` 对象的 `count` 属性后的状态

(4) 猴子甲投票事务执行以下程序代码：

```
tx.commit();
```

以上程序代码先清理缓存，根据 `Session` 缓存中 `Monkey` 对象的属性变化去同步更新数据库，最后再提交事务。同步更新数据库时执行的 `update` 语句为：

```
update MONKEYS set COUNT=1001,VERSION=1 where ID=1 and VERSION=0;
```

清理缓存时还会把缓存中 `Monkey` 对象的 `version` 属性也同步更新为 1，参见图 15-6。



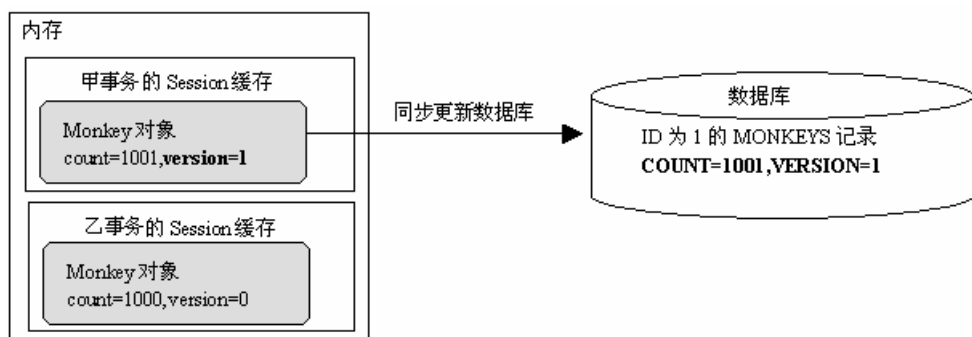


图 15-6 清理猴子甲投票事务缓存及提交猴子甲投票事务后的状态

(5) 猴子乙投票事务执行以下程序代码，修改 Session 缓存中 Monkey 对象的 count 属性，把它改为 1 001，参见图 15-7。

```
monkey.setCount(monkey.getCount()+1);
```

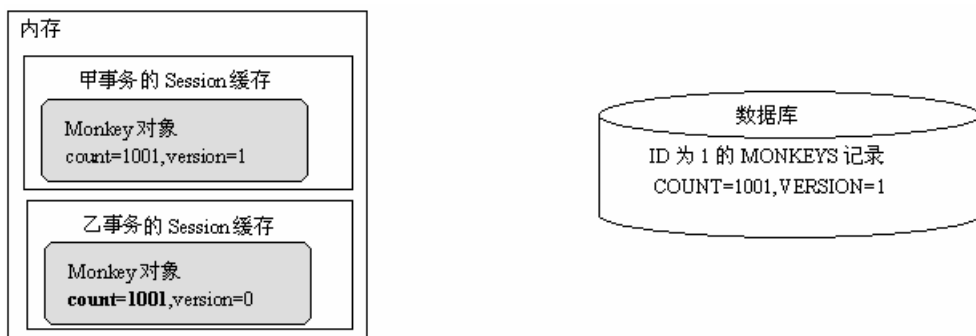


图 15-7 猴子甲投票事务修改 Session 缓存中 Monkey 对象的 count 属性后的状态

(6) 猴子乙投票事务执行以下程序代码：

```
tx.commit();
```

以上程序代码先清理缓存，试图根据 Session 缓存中 Monkey 对象的属性变化去同步更新数据库。同步更新数据库时执行的 update 语句为：

```
update MONKEYS set COUNT=1001, VERSION=1 where ID=1 and VERSION=0;
```

由于数据库中已经不存在 ID 为 1 并且 VERSION 为 0 的 MONKEYS 记录，因此以上程序代码抛出 `StaleObjectStateException`，参见图 15-8。

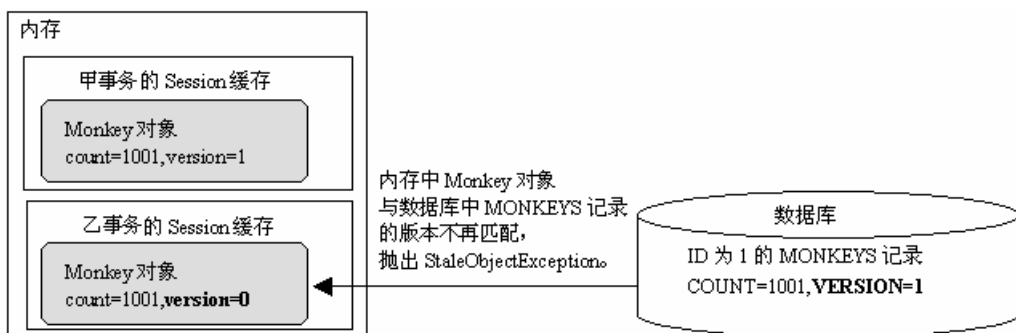


图 15-8 猴子乙投票事务抛出 StaleObjectStateException

### 15.5.2 使用<timestamp>元素

除了<version>元素，<timestamp>元素也具有同样的版本控制功能。使用<timestamp>元素的步骤如下。

(1) 在 Monkey 类中定义一个代表版本信息的属性：

```
private Date lastUpdatedTime;
public int getLastUpdatedTime() {
    return this.lastUpdatedTime;
}

public void setLastUpdatedTime(Date lastUpdatedTime) {
    this.lastUpdatedTime = lastUpdatedTime;
}
```

(2) 在 MONKEYS 表中定义一个代表版本信息的字段：

```
create table MONKEYS (
    ID bigint not null,
    NAME varchar(15),
    COUNT int,
    LAST_UPDATED_TIME timestamp,
    primary key (ID)
) type=INNODB;
```

(3) 在 Monkey.hbm.xml 文件中用<timestamp>元素来建立 Monkey 类的 lastUpdated- Time 属性与 MONKEYS 表中 LAST\_UPDATED\_TIME 字段的映射：

```
<id name="id" type="long" column="ID">
    <generator class="increment"/>
</id>
<timestamp name="lastUpdatedTime" column="LAST_UPDATED_TIME" />
.....
```

Tips

在映射文件中，<timestamp>元素必须紧跟在<id>元素的后面。

当更新一个 Monkey 对象时，Hibernate 会根据 Session 缓存中 Monkey 对象的 id 和 lastUpdatedTime 属性的当前值来定位 MONKEYS 表中的记录，如果找到匹配的记录，就更新这条记录，并且把 LAST\_UPDATED\_TIME 字段改为当前的系统时间。此外，也会把 Session 缓存中 Monkey 对象的 lastUpdatedTime 属性改为当前的系统时间。Hibernate 执行的 update 语句为：

```
update MONKEYS set NAME='智多星', COUNT=1001,
LAST_UPDATED_TIME='2010-01-27 12:01:08'
where ID=1 and LAST_UPDATED_TIME='2010-01-27 12:01:02'
```

理论上，<version>元素比<timestamp>更安全一些。数据库中 timestamp 类型表示的时间只能精确到秒，假定猴子甲投票事务在 12:01:02 100 毫秒更新 MONKEYS 表，执行的 update 语句为：

```
update MONKEYS set NAME='智多星', COUNT=1001,
LAST_UPDATED_TIME='2010-01-27 12:01:02'
where ID=1 and LAST_UPDATED_TIME='2010-01-27 12:01:02'
```

接着猴子乙投票事务在 12:01:02 500 毫秒更新 MONKEYS 表，执行的 update 语句为：

```
update MONKEYS set NAME='智多星', COUNT=1001,
LAST_UPDATED_TIME='2010-01-27 12:01:02'
where ID=1 and LAST_UPDATED_TIME='2010-01-27 12:01:02'
```

显然，猴子乙投票事务会覆盖猴子甲投票事务对票数所做的更新。此外，在集群环境中，由于各个服务器节点上的时钟可能没有同步，也会导致<timestamp>元素无法正常工作。因此，如果从头开发一个新的项目，建议采用基于整数的<version>元素。

### 15.5.3 对游离对象进行版本检查

Session 的 lock()方法显式对一个游离对象进行版本检查：

```
Session session1=sessionFactory.openSession();
tx1 = session1.beginTransaction();
Monkey monkey=(Monkey)session1.get(Monkey.class,new Long(1));
tx1.commit();
session1.close();

monkey.setCount(monkey.getCount()+1); //修改 Monkey 游离对象的属性

Session session2=sessionFactory.openSession();
tx2 = session2.beginTransaction();
```

```
session2.lock(monkey, LockMode.READ);
tx2.commit();
session2.close();
```

Session 的 lock()方法执行以下步骤。

(1) 把 Monkey 对象与当前 Session 关联。

(2) 如果设定了 LockMode.READ 模式，就比较这个 Monkey 对象的版本是否与 MONKEYS 表中对应记录的版本一致。如果不一致，说明 MONKEYS 表中对应记录已经被其他事务修改，因此会抛出 StaleObjectStateException。

表 15-11 对 Session 的 lock()方法与 update()方法进行了比较。

表 15-11 比较 Session 的 lock()方法与 update()方法		
比较两个方法	lock()方法	update()方法
不同之处	<p>如果设定了 LockMode.READ 模式，则立即进行版本检查，执行类似以下形式的 select 语句：</p> <pre>select ID from MONKEYS where ID=1 and VERSION=0;</pre> <p>如果数据库中没有匹配的记录，就抛出 StaleObjectStateException</p> <p>不会计划执行一个 update 语句</p>	<p>执行 update()方法时不会立即进行版本检查，只有当 Session 在清理缓存时，真正执行 update 语句时才会进行版本检查</p> <p>会计划执行一个 update 语句：</p> <pre>update MONKEYS set... where ID=1 and VERSION=0;</pre> <p>当 Session 清理缓存时才会执行这个 update 语句，并进行版本检查，如果数据库中没有匹配的记录，就抛出 StaleObjectStateException</p>
相似之处	都能使一个游离对象与当前 Session 关联	

## 15.6 实现乐观锁的其他方法

如果应用程序是基于已有的数据库，而数据库表中不包含代表版本或时间戳的字段，Hibernate 提供了其他实现乐观锁的办法。把<class>元素的 optimistic-lock 属性设为“all”：

```
<class name="Monkey" table="MONKEYS" optimistic-lock="all" dynamic-update="true">
```

Tips

如果把<class>元素的 optimistic-lock 属性设为“all”或者“dirty”，必须同时把 dynamic-update 属性设为 true。

Hibernate 会在 update 语句的 where 子句中包含 Monkey 对象被加载时的所有属性：

```
update MONKEYS set COUNT=1001 where ID=1 and NAME='智多星' and
COUNT=1000;
```

如果把<class>元素的 optimistic-lock 属性设为“dirty”，并且 dynamic-update 属性设为“true”：

```
<class name="Monkey" table="MONKEYS" optimistic-lock="dirty" dynamic-update="true">
```

那么，在 update 语句的 where 子句中仅包含 ID 字段及被更新过的属性：

```
update MONKEYS set COUNT=1001 where ID=1 and COUNT=1000;
```

尽管上述方法也能实现乐观锁，但是这种方法速度很慢，而且只适用于在一个 Session 中加载了对象，然后又在同一个 Session 中修改这个持久化对象的场合。以下代码在一个 Session 中加载了 Monkey 对象，然后又在另一个 Session 中更新 Monkey 对象：

```
Session session1=sessionFactory.openSession();
tx1 = session1.beginTransaction();
Monkey monkey=(Monkey)session.get(Monkey.class,new Long(1));
tx1.commit();
session1.close();

monkey.setCount(monkey.getCount()+1); //修改 Monkey 游离对象的属性

Session session2=sessionFactory.openSession();
tx2 = session2.beginTransaction();
session2.update(monkey);
tx2.commit();
session2.close();
```

第二个 Session 只能读取 Monkey 对象的所有属性的当前值，但是无法知道 Monkey 对象被第一个 Session 加载时所有属性的初始值，因此不能在 update 语句的 where 子句中包含 Monkey 对象的属性的初始值，Hibernate 执行以下 update 语句：

```
update MONKEYS set NAME='智多星',COUNT=1001 where ID=1;
```

这会导致当前事务覆盖其他事务对这条记录已做的更新。

## 15.7 小结

对于同时运行的多个事务，当这些事务访问数据库中相同的数据时，如果没有采取必要的隔离机制，就会导致各种并发问题。数据库系统采用锁来实现事务的隔离性，锁可以分为共享锁、独占锁和更新锁。许多数据库系统都有自动管理锁的功能，它们能根据事务执行的 SQL 语句，自动在保证事务间的隔离性与保证事务间的并发性之间做出权衡，然后自动为数据库资源加上适当的锁，在运行期间还会自

动升级锁的类型，以优化系统的性能。

锁机制能有效地解决各种并发问题，但是它会影响并发性能。为了能让用户根据实际应用的需要，在事务的隔离性与并发性之间做出合理的权衡，数据库系统提供了 4 种事务隔离级别供用户选择。隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。

对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为 **Read Committed**。它能够避免脏读，而且具有较好的并发性能。尽管它会导致不可重复读、虚读和第二类丢失更新这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。乐观锁通过 **Hibernate** 的版本控制功能来实现，它比悲观锁具有更好的并发性能，所以应该优先考虑使用乐观锁。

## 第 16 章 管理Session和实现对话

前面章节的范例都是由程序自主管理 Session 对象的生命周期，Session 对象的生命周期与一个事务的生命周期对应。所有访问数据库的方法都先创建一个 Session 对象，然后声明开始一个事务，当提交了事务后，就关闭 Session 对象。例如第 15 章的 15.4 节的 BusinessService 类的 vote()投票方法的主要流程如下：

```
Session session = sessionFactory.openSession();  
//创建一个 Session 对象  
session.beginTransaction(); //声明开始一个事务  
  
//具体的事务操作  
Monkey monkey=(Monkey)session.get(Monkey.class,new  
    Long(1));  
monkey.setCount(monkey.getCount()+1);  
  
session.getTransaction().commit(); //提交事务  
session.close(); //关闭 Session 对象
```

以上处理方式作为示范代码，可以很清晰地演示 Hibernate API 的用法。但对于规模庞大的实际 Java 应用，这种处理方式有以下缺点：

(1) 导致大量重复代码，在许多方法内都要重复编写创建 Session 对象和关闭 Session 对象的代码。

(2) 在多个类之间共享同一个 Session 对象比较麻烦。

为了简化 Java 应用的程序代码和软件架构，Hibernate 3 把管理 Session 对象生命周期的任务包揽到自己头上，本章介绍 Hibernate 自身提供的以下管理 Session 对象的方法：

(1) Session 对象的生命周期与本地线程绑定。

(2) Hibernate 委托程序管理 Session 对象的生命周期，值得注意的是，这种方式与本章开头讲到的由程序自主管理 Session 对象的生命周期是不一样的。

Hibernate 把包含了用户思考时间的长工作单元称为对话，本章提供实现对话的几种方式，并且分析每一种实现方式的运行性能。这些实现方式的主要区别在于 Session 对象的生命周期、事务的生命周期及清理缓存的方式不一样。

## 16.1 Hibernate管理Session对象的方式

尽管让程序自主管理 Session 对象的生命周期也是可行的，但是在实际 Java 应用中，把管理 Session 对象的生命周期交给 Hibernate，可以简化 Java 应用的程序代码和软件架构。Hibernate 自身提供了 3 种管理 Session 对象生命周期的方式：

(1) Session 对象的生命周期与本地线程绑定，参见 16.2 节。

(2) Session 对象的生命周期与 JTA 事务绑定，本书未对此做介绍，在作者的另一本书《精通 Hibernate：Java 对象持久化技术详解》中介绍了这种方式。

(3) Hibernate 委托程序管理 Session 对象的生命周期，参见 16.4 节。

### Tips

为了叙述的方便，下文有时把管理 Session 对象的生命周期简称为管理 Session。

对于本书第 15 章的 15.4 节提到的 **BusinessService** 类，它包含了业务逻辑代码，创建 **SessionFactory** 对象的代码、管理 Session 对象的代码，以及执行数据库事务的代码。对于复杂的软件应用，需要细分 **BusinessService** 类的功能，建立精粒度的对象模型。精粒度的对象模型把功能逐步分解为多个模块，然后分配给多个类来协作完成。精粒度的对象模型可以减少重复代码，而且能提高每个类的独立性，有利于软件的维护和可重用。本节将进一步细分 **BusinessService** 类的功能，并且由多个类来分担 **BusinessService** 类的功能：

- **HibernateUtil** 类：实用类，负责创建一个应用范围内的 **SessionFactory** 对象，它的 `getSessionFactory()` 静态方法返回这个 **SessionFactory** 对象。它的 `getCurrentSession()` 静态方法返回当前的 Session 对象。
- **Monkey** 类：持久化类，代表猴子。
- **MonkeyDAO** 类：封装了通过 **Hibernate** API 来访问数据库，进行查询及更新 **Monkey** 对象的代码。不负责声明数据库事务。**DAO** (Data Access Object) 表示数据访问对象。
- **BusinessService** 类：负责处理投票业务，通过 **MonkeyDAO** 类来查询及更新 **Monkey** 对象，数据库事务的声明由 **BusinessService** 类来完成。

图 16-1 显示了这些类之间的依赖关系。



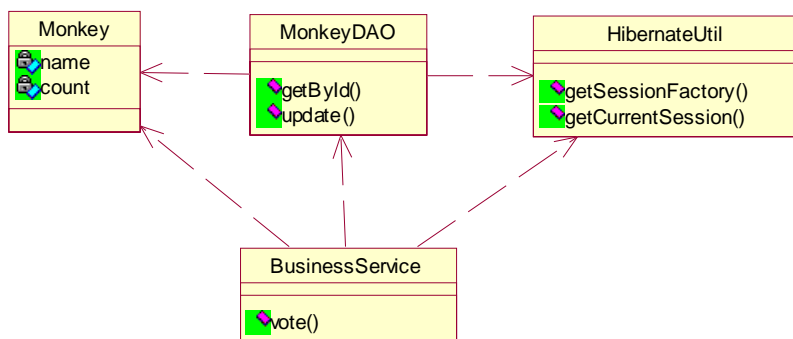


图 16-1 精粒度的对象模型

HibernateUtil 类是一个实用类，它的源代码参见例程 16-1。

#### 例程 16-1 HibernateUtil.java

```

package mypack;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static SessionFactory sessionFactory;

    static{
        try {
            //创建被整个 Java 应用共享的 SessionFactory 对象
            sessionFactory=new Configuration()
                                .configure()
                                .buildSessionFactory();
        }catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static Session getCurrentSession(){
        return sessionFactory.getCurrentSession();
        //返回当前的 Session 对象
    }
}
  
```

SessionFactory 对象是重量级对象，在一个 Java 应用中，对于一个数据库存储

源,只需要创建一个代表该存储源的 `SessionFactory` 对象,它被整个 Java 应用共享。

`MonkeyDAO` 类封装了与 `Monkey` 对象有关的访问数据库的代码,参见例程 16-2。`MonkeyDAO` 类通过 `HibernateUtil` 类的 `getCurrentSession()` 方法来得到当前的 `Session` 对象。`MonkeyDAO` 类的 `getById()` 及 `update()` 方法既不用管理 `Session`,也不用声明事务。

例程 16-2 `MonkeyDAO.java`

```
package mypack;

public class MonkeyDAO{
    public Monkey getById(long id){
        return (Monkey)HibernateUtil.getCurrentSession()
            .get(Monkey.class,new Long(id));
    }

    public void update(Monkey monkey){
        HibernateUtil.getCurrentSession().saveOrUpdate(monkey);
    }
}
```

Hibernate 到底如何管理 `Session` 对 Java 应用是透明的。无论是那种管理方式,`MonkeyDAO` 类都不必自己创建 `Session` 对象,只需调用 `HibernateUtil` 类的 `getCurrentSession()` 方法,就能获得当前的 `Session` 对象。

而 `HibernateUtil` 类的 `getCurrentSession()` 方法实际上是调用 `SessionFactory` 对象的 `getCurrentSession()` 方法,来获得当前的 `Session` 对象。由此可见,Hibernate 内部封装了管理 `Session` 对象的生命周期的实现细节。

当 Java 应用改变 Hibernate 的 `Session` 管理方式时,无须修改 `Monkey` 类、`MonkeyDAO` 类和 `HibernateUtil` 类的源代码,只需修改 Hibernate 的配置文件及 `BusinessService` 类就行了。在 Hibernate 的配置文件中,`hibernate.current_session_context_class` 属性用于指定 `Session` 管理方式,可选值包括:

- `thread`: `Session` 对象的生命周期与本地线程绑定。
- `jta`: `Session` 对象的生命周期与 JTA 事务绑定。
- `managed`: Hibernate 委托程序来管理 `Session` 对象的生命周期。

## 16.2 Session对象的生命周期与本地线程绑定

如果把 Hibernate 配置文件的 `hibernate.current_session_context_class` 属性设为 `thread`,Hibernate 就会按照与本地线程绑定的方式来管理 `Session` 对象的生命周期。例程 16-3 为这种方式下 `BusinessService` 类的源代码。

例程 16-3 BusinessService.java

```

package mypack;
public class BusinessService{
    private MonkeyDAO md=new MonkeyDAO();

    public void vote(long monkeyId){
        try {
            //声明开始事务
            HibernateUtil.getCurrentSession().beginTransaction();

            Monkey monkey=md.getById(monkeyId);
            monkey.setCount(monkey.getCount()+1);

            //提交事务
            HibernateUtil.getCurrentSession().getTransaction().commit();

        }catch (RuntimeException e) {
            try{
                //撤销事务
                HibernateUtil.getCurrentSession().getTransaction().
                    rollback();
            }catch(RuntimeException ex){
                ex.printStackTrace();
            }
            throw e;
        }
    }

    public static void main(String args[]) {
        new BusinessService().vote(1);
    }
}

```

当运行 BusinessService 类的 vote()方法时, 该 vote()方法及 MonkeyDAO 对象的 getById()方法实际上都通过 SessionFactory 对象的 getCurrentSession()方法来获得当前的 Session 对象。当前的 Session 对象何时创建及何时被销毁由 Hibernate 来管理。

### Tips

MonkeyDAO 类及 BusinessService 类都没有通过 import 语句引入 Hibernate API 中的任何接口和类, 从而提高了这些类的独立性。通过 SessionFactory 对象的 getCurrentSession()方法来获得当前 Session 对象的代码封装在 HibernateUtil 类中。

Hibernate 按照以下规则把 Session 与本地线程绑定:

- 当一个线程 (假定为线程 A) 第一次调用 SessionFactory 对象的 getCurrentSession()方法时, 该方法会创建一个新的 Session 对象 (假定为

SessionA 对象), 把它与线程 A 绑定, 并将 SessionA 对象返回。

- 接下来, 当线程 A 再次调用 SessionFactory 对象的 getCurrentSession()方法时, 该方法始终返回 SessionA 对象。
- 当线程 A 提交与 SessionA 对象关联的事务时, Hibernate 会自动清理 SessionA 对象的缓存, 然后在提交事务后, 关闭 SessionA 对象。此外, 当线程 A 撤销与 SessionA 对象关联的事务时, 也会自动关闭 SessionA 对象。
- 接下来如果线程 A 再次调用 SessionFactory 对象的 getCurrentSession()方法, 该方法又会创建一个新的 Session 对象 (假定为 SessionB 对象), 把它与线程 A 绑定, 并将 SessionB 对象返回。

本书范例位于配套光盘的 sourcecode/chapter16 目录下。运行 BusinessService 类时, 其主线程执行 BusinessService 类的 vote()方法, BusinessService 类的 vote()方法还调用了 MonkeyDAO 类的 getById()方法。图 16-2 为主线程执行 BusinessService 类的 vote()方法的时序图。

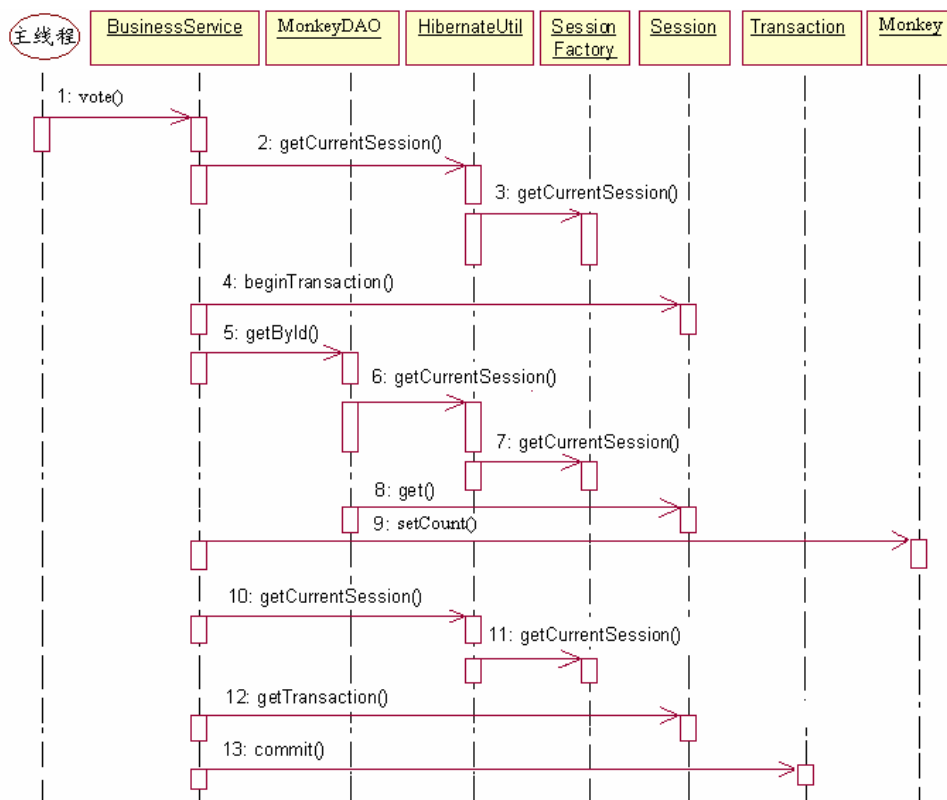


图 16-2 主线程执行 BusinessService 类的 vote()方法的时序图

从图 16-2 可以看出, 主线程执行步骤 3、步骤 7 和步骤 11 时, 都调用了

SessionFactory 对象的 getSession()方法，其中步骤 3 创建一个新的 Session 对象，这个 Session 对象与主线程绑定，接下来步骤 7 和步骤 11 都返回这个 Session 对象。步骤 13 提交事务，此时 Hibernate 会自动关闭与主线程绑定的 Session 对象。

## 16.3 实现对话

在本书中，对话是指包含了用户思考时间的长工作单元。假定把实际应用中的投票事务看做一个对话，图 16-3 显示了用户与应用程序之间的交互过程。对话与数据库事务的区别在于：在对话过程中会包含并不涉及操纵数据库的操作，并且这种操作会占用很长时间，如等待用户输入特定数据的操作就可能会花去很长时间，这完全取决于用户的思考时间。

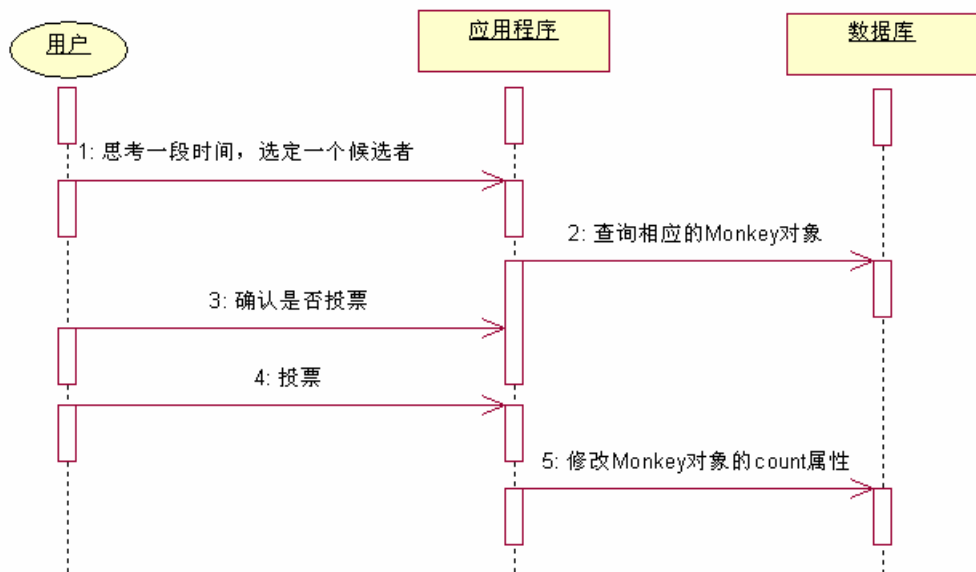


图 16-3 用户与应用程序之间的交互过程

为了保证对话顺利进行，应用程序在实现对话时必须满足以下两个要求：

(1) 对话中的数据保持一致，即在多用户并发访问环境下不出现任何并发问题。

(2) 对话和事务一样，也具有原子性。整个对话中的数据库操作要么全部提交，要么全部撤销。

到底如何实现对话呢？最简单的做法是把整个对话看做一个数据库事务，并且与一个 Session 对象对应。按照这种方式实现的投票对话过程的演示代码如下：

```
//声明开始事务
HibernateUtil.getCurrentSession().beginTransaction();
```

```

System.out.println("请输入候选者的 ID: ");
//等待用户输入候选者的 ID, 此操作可能会花去很长时间, 取决于用户的思考时间
long monkeyId=new Scanner(System.in).nextLong();

Monkey monkey=md.getById(monkeyId);
System.out.println("候选者的当前票数为: "+monkey.getCount());

System.out.println("您确信要投票吗(Y/N): ");
//等待用户确认是否投票, 此操作可能会花去很长时间, 取决于用户的思考时间
String flag=new Scanner(System.in).next();
if(flag.equals("N"))return;

monkey.setCount(monkey.getCount()+1);

//提交事务, 在 Session 与本地线程绑定的方式下, 会自动关闭 Session 对象
HibernateUtil.getCurrentSession().getTransaction().commit();
System.out.println("投票成功, 候选者的当前票数为: "+monkey.getCount());

```

以上实现方式很容易满足对话的两个要求:

(1) 采用乐观锁来解决并发问题。本书第 15 章的 15.5 节已经介绍了 Hibernate 利用版本控制机制来实现乐观锁的方法。

(2) 由于整个对话就是一个数据库事务, 因此数据库层会保证事务的原子性。

但用这种方式来实现对话会大大降低程序的运行性能, 原因在于:

(1) 长时间打开一个 Session 对象。一方面 Session 对象的缓存需要长时间占用内存, 另一方面, Session 对象还会长时间占用数据库连接。假如一个 Java 应用同时打开了很多对话, 每个对话都对应一个 Session 对象, 这些 Session 对象会消耗大量资源。

(2) 整个对话对应一个数据库事务, 导致这个数据库事务会长时间占用数据库的相关资源, 从而降低数据库的运行性能。

为了解决上述问题, Hibernate 提供了两种更好的实现对话的方式:

(1) 使用游离对象: 一个对话包括多个短事务, 并且每个短事务对应一个单独的 Session 对象。事务之间通过游离对象传递业务数据。

(2) 使用手工清理缓存模式下的 Session 对象: 一个对话包括多个短事务, 并且整个对话对应一个 Session 对象。

下文仍然以投票对话为例, 介绍以上两种实现方式的特点和利弊。

### 16.3.1 使用游离对象

以下程序代码把投票对话中包含用户思考时间的操作从事务中分离出来。投票对话由两个短事务构成：查询候选者事务和修改票数事务。每个事务对应一个 Session 对象。本节的范例代码采用 Session 与本地线程绑定的方式。

```
System.out.println("请输入候选者的 ID: ");
//等待用户输入候选者的 ID, 此操作可能会花去很长时间, 取决于用户的思考时间
long monkeyId=new Scanner(System.in).nextLong();

//创建一个 Session 对象, 声明开始查询候选者事务
HibernateUtil.getCurrentSession().beginTransaction();
Monkey monkey=md.getById(monkeyId);
//提交事务, 在 Session 与本地线程绑定的方式下, 会自动关闭 Session 对象
HibernateUtil.getCurrentSession().getTransaction().commit();

System.out.println("候选者的当前票数为: "+monkey.getCount());

System.out.println("您确信要投票吗(Y/N): ");
//等待用户确认是否投票, 此操作可能会花去很长时间, 取决于用户的思考时间
String flag=new Scanner(System.in).next();
if(flag.equals("N"))return;

//monkey 为游离对象
monkey.setCount(monkey.getCount()+1);

//创建一个 Session 对象, 声明开始修改票数事务
HibernateUtil.getCurrentSession().beginTransaction();
md.update(monkey);
//提交事务, 在 Session 与本地线程绑定的方式下, 会自动关闭 Session 对象
HibernateUtil.getCurrentSession().getTransaction().commit();

System.out.println("投票成功, 候选者的当前票数为: "+monkey.getCount());
```

以上两个短事务不包括用户思考时间, 因此事务操作时间很短, 不会长时间占用数据库资源。并且生命周期与短事务对应的 Session 对象也可以及时释放缓存和数据库连接。所以这种实现对话的方式可以提高程序的运行性能。

以下例程 16-4 的 BusinessService1 类按这种方式实现了投票对话。投票对话中的查询候选者及修改票数事务分别由 getMonkey() 和 updateMonkey() 方法实现。vote() 方法中包括了与用户交互的操作, 并且先后调用了 getMonkey() 和 updateMonkey() 方法。

例程 16-4 BusinessService1.java

```
package mypack;
```

```
import java.io.*;
import java.util.Scanner;

public class BusinessService1{
    private MonkeyDAO md=new MonkeyDAO();

    public void vote()throws Exception{
        System.out.println("请输入候选者的 ID: ");
        //等待用户输入候选者的 ID, 此操作可能会化去很长时间, 取决于用户的思考时间
        long monkeyId=new Scanner(System.in).nextLong();

        Monkey monkey=getMonkey(monkeyId);

        System.out.println("候选者的当前票数为: "+monkey.getCount());

        System.out.println("您确信要投票吗(Y/N): ");
        //等待用户确认是否投票, 此操作可能会花去很长时间, 取决于用户的思考时间
        String flag=new Scanner(System.in).next();
        if(flag.equals("N"))return;

        monkey.setCount(monkey.getCount()+1);
        updateMonkey(monkey);

        System.out.println("投票成功, 候选者的当前票数为: "+monkey.getCount());
    }

    public Monkey getMonkey(long monkeyId){
        try{
            //创建一个 Session 对象, 声明开始查询候选者事务
            HibernateUtil.getCurrentSession().beginTransaction();

            Monkey monkey=md.getById(monkeyId);

            //提交事务, 在 Session 与本地线程绑定的方式下, 会自动关闭 Session 对象
            HibernateUtil.getCurrentSession().getTransaction().commit();

            return monkey;
        }catch (RuntimeException e) {
            try{
                //撤销事务
                HibernateUtil.getCurrentSession().getTransaction().
                    rollback();
            }catch(RuntimeException ex){
                ex.printStackTrace();
            }
        }
    }
}
```



```

        throw e;
    }

}

public void updateMonkey(Monkey monkey){
    try{
        //创建一个 Session 对象, 声明开始修改票数事务
        HibernateUtil.getCurrentSession().beginTransaction();

        md.update(monkey);

        //提交事务, 在 Session 与本地线程绑定的方式下, 会自动关闭 Session 对象
        HibernateUtil.getCurrentSession().getTransaction().commit();

    }catch (RuntimeException e) {
        try{
            //撤销事务
            HibernateUtil.getCurrentSession().getTransaction().
                rollback();
        }catch(RuntimeException ex){
            ex.printStackTrace();
        }
        throw e;
    }
}

public static void main(String args[]) throws Exception {
    new BusinessService1().vote();
}
}

```

这种实现对话的方式为了满足对话的第一个要求, 可以采用乐观锁来解决并发问题。但是这种方式要满足对话的第二个要求, 即保证对话的原子性有些麻烦, 因为一个对话被分成了多个数据库事务, 并且每个事务对应一个单独的 **Session** 对象。如果撤销其中一个事务, 无法自动撤销对话中已经提交的其他事务。假定对话由  $n$  个事务组成, 依次为事务 1、事务 2……事务  $n$ 。要保证对话的原子性, 目前有两种办法可选:

(1) 把与事务 1 至事务  $n-1$  对应的 **Session** 对象的清理缓存模式设为 **FlushMode.MANUAL**, 这样, 当提交事务 1 至事务  $n-1$  时, 均不会自动清理缓存, 因此不会对数据库有任何更新操作。仅把与事务  $n$  对应的 **Session** 对象的清理缓存模式设为默认的 **FlushMode.AUTO**。当提交事务  $n$  时, 会自动清理缓存, 根据缓存中对象的变化来同步更新数据库。

(2) 在程序中提供补偿代码, 手工撤销已经提交的事务。假定已经提交了事务 1、事务 2 和事务 3, 但需要撤销事务 4, 那么撤销事务 4 时, 必须通过补偿代码手工撤销事务 1、事务 2 和事务 3 对数据库所做的更新。

### 16.3.2 使用手工清理缓存模式下的Session

采用这种方式, 整个对话由多个短事务构成, 并且整个对话对应一个 Session 对象, 这个 Session 对象的生命周期由程序自主管理。程序处理对话的主要流程如下:

(1) 创建完 Session 对象后, 立即调用 `session.setFlushMode(FlushMode.MANUAL)` 方法, 把缓存模式设为手工清理模式。

(2) 在手工清理缓存模式下, 当程序声明提交事务时, **Hibernate** 不会自动清理缓存, 因此不会同步更新数据库; 不过, **Hibernate** 会自动释放 Session 对象占用的数据库连接。

(3) 当程序声明开始事务时, 假如当前 Session 对象不占用数据库连接, **Hibernate** 会自动为它分配数据库连接。

(4) 只有当对话快结束, 在提交最后一个事务之前, 程序需要调用 `session.flush()` 方法手工清理缓存, 根据缓存中对象的变化去同步更新数据库。

以下例程 16-5 的 **BusinessService2** 的 `vote()` 方法创建了一个 Session 对象, 它的声明周期与整个投票对话的生命周期对应。

例程 16-5 BusinessService2.java

```
package mypack;
import java.io.*;
import java.util.Scanner;
import org.hibernate.*;

public class BusinessService2{
    private MonkeyDAO2 md=new MonkeyDAO2();

    public void vote()throws Exception{
        Session session=null;
        try{
            System.out.println("请输入候选者的 ID: ");
            //等待用户输入候选者的 ID, 此操作可能会化去很长时间, 取决于用户的思考时间
            long monkeyId=new Scanner(System.in).nextLong();

            //创建一个 Session 对象, 由程序自主管理 Session 对象的生命周期
            session=HibernateUtil.getSessionFactory().openSession();
            //设为手工清理缓存模式
            session.setFlushMode(FlushMode.MANUAL);
        }
    }
}
```

```

//声明开始查询候选者事务
session.beginTransaction();
Monkey monkey=md.getById(monkeyId,session);
//提交查询候选者事务,释放 session 占用的数据库连接
session.getTransaction().commit();

System.out.println("候选者的当前票数为: "+monkey.getCount());

System.out.println("您确信要投票吗(Y/N): ");
//等待用户确认是否投票,此操作可能会花去很长时间,取决于用户的思考时间
String flag=new Scanner(System.in).next();
if(flag.equals("N"))return;

monkey.setCount(monkey.getCount()+1);

//声明开始修改票数事务,为 session 重新分配数据库连接
session.beginTransaction();
md.update(monkey,session);

//清理缓存
session.flush();

//提交修改票数事务
session.getTransaction().commit();

System.out.println("投票成功,候选者的当前票数为:
    "+monkey.getCount());
}catch (RuntimeException e) {
    try{
        //撤销事务
        session.getTransaction().rollback();
    }catch(RuntimeException ex){
        ex.printStackTrace();
    }
    throw e;
}finally{
    session.close();
}
}

public static void main(String args[]) throws Exception {
    new BusinessService2().vote();
}
}

```

以上两个事务不包括用户思考时间，因此事务操作时间很短，不会长时间占用数据库资源。**Session** 对象的生命周期尽管很长，与整个对话对应，但是由于每次提交事务时都会自动释放数据库连接，所以这种实现对话的方式也具有较好的运行性能。

这种实现对话的方式为了满足对话的第一要求，可以采用乐观锁来解决并发问题。这种方式也很容易满足对话的第二个要求，即保证对话的原子性。尽管整个对话被分成了多个事务，但是仅在提交最后一个事务之前才清理 **Session** 对象的缓存，所以实际上只有最后一个事务才会更新数据库。如果撤销最后一个事务，实际上就撤销了整个对话。

本方式的一个弱点在于 **Session** 对象的缓存需要长时间占用内存。因此，如果在对话中 **Session** 对象的缓存需要存放大量持久化对象，则不适合采用这种对话实现方式。

## 16.4 Hibernate 委托程序来管理 Session

16.3.2 节的例程 16-5 的 **BusinessService2** 类完全由程序来自主管理 **Session** 对象的生命周期，何时创建 **Session**、何时清理缓存及何时关闭 **Session**，都由程序来负责。**BusinessService2** 类如果要和 **MonkeyDAO2** 类共享 **Session** 对象，就必须通过方法参数来传递 **Session** 对象。例如，**BusinessService2** 类调用 **MonkeyDAO2** 类的 **getById()** 方法时，需要把当前的 **Session** 对象传给 **getById()** 方法。**getById()** 方法的定义如下：

```
public Monkey getById(long id, Session session){
    return (Monkey)session.get(Monkey.class, new Long(id));
}
```

如果 **MonkeyDAO2** 类也可以通过 **SessionFactory** 的 **getCurrentSession()** 方法来获得当前 **Session** 对象，就可以简化程序的结构。为了达到这一目标，**Hibernate** 允许委托程序来管理 **Session**，步骤如下。

(1) 把 **Hibernate** 配置文件的 **hibernate.current\_session\_context\_class** 属性设为 **managed**。

(2) 当程序创建 **Session** 对象后或者声明开始一个事务前，调用 **org.hibernate.context.ManagedSessionContext** 类的 **bind()** 方法，该方法把 **Session** 对象与当前线程绑定。

(3) 在一个事务中，程序可通过 **SessionFactory** 的 **getCurrentSession()** 方法来获得当前 **Session** 对象。

(4) 当程序提交对话中的一个事务之前, 调用 `org.hibernate.context.ManagedSessionContext` 类的 `unbind()` 方法, 该方法解除当前 Session 对象与当前线程的绑定。

`org.hibernate.context.ManagedSessionContext` 类提供了以下静态方法:

- `bind(Session session)`: 把 Session 与当前线程绑定。
- `unbind(SessionFactory factory)`: 解除 Session 对象与当前线程的绑定。
- `hasBind(SessionFactory factory)`: 判断是否存在与当前线程绑定的 Session 对象。

### Tips

`ManagedSessionContext` 类的 `unbind(SessionFactory factory)` 方法通过参数 `factory` 对象的 `getCurrentSession()` 方法得到当前的 Session 对象, 然后解除 Session 对象与当前线程的绑定。

例程 16-6 的 `BusinessService3` 类按照上述方式重现实现了投票对话。

例程 16-6 `BusinessService3.java`

```
package mypack;
import java.io.*;
import java.util.Scanner;
import org.hibernate.classic.Session;
import org.hibernate.context.ManagedSessionContext;
import org.hibernate.FlushMode;

public class BusinessService3{
    private MonkeyDAO md=new MonkeyDAO();

    public void vote()throws Exception{
        Session session=null;
        try{
            System.out.println("请输入候选者的 ID: ");
            //等待用户输入候选者的 ID, 此操作可能会化去很长时间, 取决于用户的思考时间
            long monkeyId=new Scanner(System.in).nextLong();

            //创建一个 Session 对象, 由程序自主管理 Session 对象的生命周期
            session=HibernateUtil.getSessionFactory().openSession();
            //设为手工清理缓存模式
            session.setFlushMode(FlushMode.MANUAL);
            ManagedSessionContext.bind(session);

            //声明开始查询候选者事务
            session.beginTransaction();
            Monkey monkey=md.getById(monkeyId);

            ManagedSessionContext.unbind(HibernateUtil.
                getSessionFactory());
```

```

        //提交查询候选者事务,释放 Session 占用的数据库连接
        session.getTransaction().commit();
        System.out.println("候选者的当前票数为: "+monkey.getCount());

        System.out.println("您确信要投票吗(Y/N): ");
        //等待用户确认是否投票,此操作可能会花去很长时间,取决于用户的思考时间
        String flag=new Scanner(System.in).next();
        if(flag.equals("N"))return;

        monkey.setCount(monkey.getCount()+1);

        ManagedSessionContext.bind(session);

        //声明开始修改票数事务,为 Session 重新分配数据库连接
        session.beginTransaction();
        md.update(monkey);

        ManagedSessionContext.unbind(HibernateUtil.
            getSessionFactory());

        //清理缓存
        session.flush();

        //提交修改票数事务
        session.getTransaction().commit();

        System.out.println("投票成功,候选者的当前票数为:
            "+monkey.getCount());
    }catch (RuntimeException e) {
        try{
            //撤销事务
            session.getTransaction().rollback();
        }catch(RuntimeException ex){
            ex.printStackTrace();
        }
        throw e;
    }finally{
        session.close();
    }
}

public static void main(String args[]) throws Exception {
    new BusinessService3().vote();
}
}

```

图 16-4 演示了投票对话中 Session 对象及两个事务的生命周期。MSC 为 ManagedSessionContext 的简写。

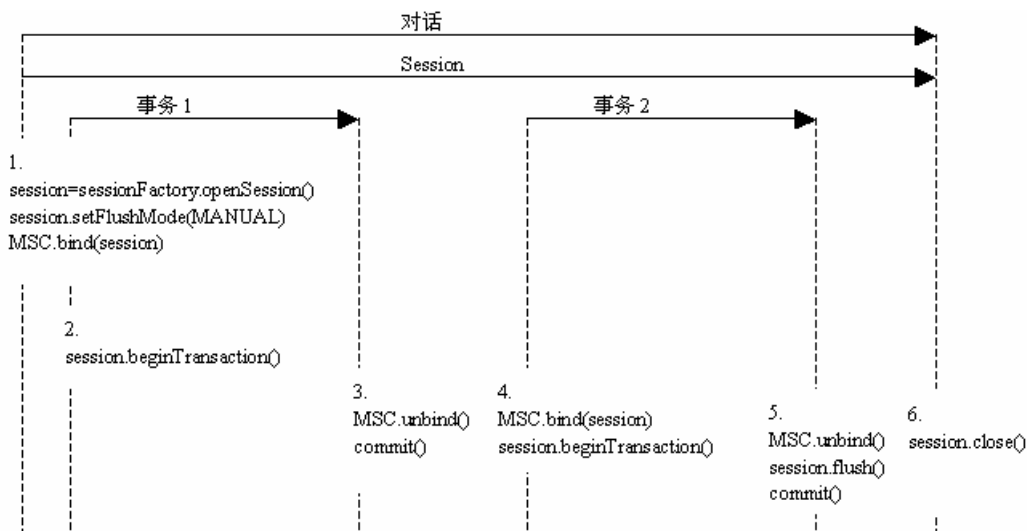


图 16-4 投票对话中 Session 对象及两个事务的生命周期

## 16.5 小结

归纳起来，管理 Session 对象的生命周期有 4 种方式。第一种方式为完全由程序自主管理 Session，其余 3 种方式均由 Hibernate 来管理 Session，程序可通过 SessionFactory 的 getCurrentSession() 方法来获得当前的 Session 对象。在 Hibernate 的配置文件中，hibernate.current\_session\_context\_class 属性用于指定 Hibernate 管理 Session 的方式，可选值包括：

- thread: Session 对象的生命周期与本地线程绑定。
- jta: Session 对象的生命周期与 JTA 事务绑定。
- managed: Hibernate 委托程序来管理 Session 对象的生命周期。

本章介绍了 thread 和 managed 这两种管理 Session 方式的用法，表 16-1 总结了这两种方式的特点。

表 16-1 Hibernate 管理 Session 的方式

管理 Session 的方式	何时创建 Session	何时关闭 Session	程序如何获得 Session
thread	当一个线程调用 SessionFactory 的 getCurrentSession() 方法，而此时没有 Session 与当前线程绑定时，Hibernate 自动创建 Session 对象	当一个线程提交或撤销事务后，Hibernate 自动关闭 Session 对象	调用 SessionFactory 的 getCurrentSession() 方法

(续表)

管理 Session 的方式	何时创建 Session	何时关闭 Session	程序如何获得 Session
managed	由程序决定何时创建 Session。当程序创建 Session 对象后或者声明开始一个事务前,调用 ManagedSessionContext 类的 bind()方法,该方法把 Session 对象与当前线程绑定	由程序决定何时关闭 Session。当程序提交一个事务之前,调用 ManagedSessionContext 类的 unbind()方法,该方法解除 Session 对象与当前线程的绑定	

归纳起来,对话有 3 种实现方式,表 16-2 对这 3 种实现方式做了归纳。

表 16-2 对话的 3 种实现方式			
对话实现方式	方式一: 整个对话对应一个事务	方式二: 使用游离对象	方式三: 使用手工清理缓存模式下的 Session 对象
特点	整个对话对应一个 Session 对象,并且对应一个长事务	整个对话对应多个 Session 对象,并且每个 Session 对象对应一个短事务	整个对话对应一个 Session 对象,并且这个 Session 对象对应多个短事务
提高性能的优势		由于每个 Session 对象的生命周期与短事务对应,因此 Session 对象不会长时间占用内存和数据库连接;此外,数据库事务也不会长时间占用数据库的相关资源	每次提交事务时,都会自动释放 Session 对象占用的数据库连接;此外,数据库事务也不会长时间占用数据库的相关资源
降低性能的劣势	Session 对象的缓存需要长时间占用内存; Session 对象会长时间占用数据库连接;数据库事务会长时间占用数据库的相关资源	对话中的多个 Session 对象可能会重复访问数据库,到数据库中加载一些相同数据	Session 对象的缓存需要长时间占用内存
如何保证对话的原子性	由单个数据库事务来保证对话的原子性	方法一: 仅在对话的最后一个事务中才更新数据库;方法二: 提供补偿代码,手工撤销已经提交的事务	仅在对话的最后一个事务中才更新数据库
如何避免并发问题	采用乐观锁来解决并发问题		

本章范例程序位于配套光盘的 sourcecode\chapter16 目录下。运行该程序之前,需要先在 SAMPLEDB 数据库中手工创建 MONKEYS 表,然后加入测试数据,相关的 SQL 脚本文件为 chapter16\schema\sampledb.sql。在 DOS 下分别输入命令“ant run”、“ant run1”、“ant run2”和“ant run3”,它们将分别运行 BusinessService 类、BusinessService1 类、BusinessService2 类和 BusinessService3 类。在运行 BusinessService3 类时,要确保 hibernate.cfg.xml 配置文件中 hibernate.current\_session\_context\_class 属性为“managed”。例如运行 BusinessService2 类时,用户将与程序展开以下对话:



请输入候选者的 ID:

1

候选者的当前票数为: 1000

您确信要投票吗 (Y/N):

Y

投票成功, 候选者的当前票数为: 1001

9

10

11

12

13

14

15

16